Universidad Carlos III de Madrid
Escuela Politécnica Superior
Computer Science Department

Doctoral Thesis

A multi-tier cached I/O architecture for massively parallel supercomputers

Author: **Francisco Javier García Blas**
Advisors: **Florin Isaila**
Jesús Carretero Pérez

Leganés, 2010

Universidad Carlos III de Madrid
Escuela Politécnica Superior
Departamento de Informática

Tesis Doctoral

A multi-tier cached I/O architecture for massively parallel supercomputers

Autor: **Francisco Javier García Blas**
Directores: **Florin Isaila**
Jesús Carretero Pérez

Leganés, 2010

TESIS DOCTORAL

A multi-tier cached I/O architecture for massively parallel supercomputers

AUTOR: **Francisco Javier García Blas**
DIRECTORES: **Florin Isaila**
Jesús Carretero Pérez

TRIBUNAL CALIFICADOR

PRESIDENTE:

VOCAL:

VOCAL:

VOCAL:

SECRETARIO:

CALIFICACION:

Leganés, a de de 2010

Acknowledgments

Many people with encouragement, advice, friendship and affection, have contributed to making this thesis come to fruition. This thesis is partly yours too, receive my cordial thanks.

I would like to gratefully acknowledge the enthusiastic supervision of Dr. Florin Isaila, co-advisor of this thesis. Thank you for your excellent work, for your advice, for your experience, for your continued support and confidence you've always placed in me. I am also grateful for his patience with all the problems which rinse during the elaboration of this thesis. Furthermore, I am thankful to you for being not only a advisor but also a good friend.

A special thanks goes to my co-advisor, Dr. Jesús Carretero. Your support and confidence in my work and ability to guide my research has been an invaluable contribution not only in developing this thesis, but also in my training. His enthusiasm and direction kept this project going. Her leadership, support, attention to detail, hard work have set an example I hope to match some day.

The remaining colleges in ARCOS research group of University Carlos III of Madrid, with whom, during these five years, I had the pleasure of working (Félix, Jose Daniel, David, Alejandro, Javi, Cristina, Luis Miguel, Soledad, Borja, Alejandra, Laura, and Fortran). Your help has been invaluable.

I am also pleased to thank the support in different venues where I spent part of my PhD; a special word of gratitude goes to Alexander Schulz and Rainer Keller, for hosting me in Stuttgart. Special thanks Rainer for your revision and suggestions. Also, remember with gratitude the months spent in Chicago with Rob Latham and Rob Ross.

Words fail me to express my appreciation to Penélope, whose dedication, love ,and persistent confidence in me, has taken the load off my shoulder. And, of course, thank my parents and brother for help this thesis succeed and, above all, for so many other things. Many thanks to all my friends who have been around these four years, specially Alfonso, who has been helping me at different occasions.

During the course of this work, I was supported in part by Spanish Ministry of Education under project TEALES (TIN 2007-63092), by the U.S. Dept. of Energy under Contracts DE-FC02-07ER25808, DE-FC02-01ER25485, and DE-AC02-06CH11357, and NSF HECURA CCF-0621443, NSF SDCIOCI-0724599, and NSF ST-HEC CCF-0444405. This work was also supported in part by DOE SCIDAC-2: Scientific Data Management Center for Enabling Technologies (CET) grant DE-FC02-07ER25808, DE-FC02-01ER25485, NSF HECURA CCF-0621443, NSF SDCI OCI-0724599, and NSF ST-HECCCF-0444405.

Francisco Javier García Blas.

Abstract

Recent advances in storage technologies and high performance interconnects have made possible in the last years to build, more and more potent storage systems that serve thousands of nodes. The majority of storage systems of clusters and supercomputers from Top 500 list are managed by one of three scalable parallel file systems: GPFS, PVFS, and Lustre.

Most large-scale scientific parallel applications are written in Message Passing Interface (MPI), which has become the de-facto standard for scalable distributed memory machines. One part of the MPI standard is related to I/O and has among its main goals the portability and efficiency of file system accesses. All of the above mentioned parallel file systems may be accessed also through the MPI-IO interface.

The I/O access patterns of scientific parallel applications often consist of accesses to a large number of small, non-contiguous pieces of data. For small file accesses the performance is dominated by the latency of network transfers and disks. Parallel scientific applications lead to interleaved file access patterns with high interprocess spatial locality at the I/O nodes. Additionally, scientific applications exhibit repetitive behaviour when a loop or a function with loops issues I/O requests. When I/O access patterns are repetitive, caching and prefetching can effectively mask their access latency. These characteristics of the access patterns motivated several researchers to propose parallel I/O optimizations both at library and file system levels. However, these optimizations are not always integrated across different layers in the systems.

In this dissertation we propose a novel generic parallel I/O architecture for clusters and supercomputers. Our design is aimed at large-scale parallel architectures with thousands of compute nodes. Besides acting as middleware for existing parallel file systems, our architecture provides on-line virtualization of storage resources. Another objective of this thesis is to factor out the common parallel I/O functionality from clusters and supercomputers in generic modules in order to facilitate porting of scientific applications across these platforms.

Our solution is based on a multi-tier cache architecture, collective I/O, and asynchronous data staging strategies hiding the latency of data transfer between cache tiers. The thesis targets to reduce the file access latency perceived by the data-intensive parallel scientific applications by multi-layer asynchronous data transfers. In order to accomplish this objective, our techniques leverage the multi-core architectures by overlapping computation with communication and I/O in parallel threads.

Prototypes of our solutions have been deployed on both clusters and Blue Gene supercomputers. Performance evaluation shows that the combination of collective strategies with overlapping of computation, communication, and I/O may bring a substantial performance benefit for access patterns common for parallel scientific applications.

Resumen

En los últimos años se ha observado un incremento sustancial de la cantidad de datos producidos por las aplicaciones científicas paralelas y de la necesidad de almacenar estos datos de forma persistente. Los sistemas de ficheros paralelos como PVFS, Lustre y GPFS han ofrecido una solución escalable para esta demanda creciente de almacenamiento.

La mayoría de las aplicaciones científicas son escritas haciendo uso de la interfaz de paso de mensajes (MPI), que se ha convertido en un estándar de-facto de programación para las arquitecturas de memoria distribuida. Las aplicaciones paralelas que usan MPI pueden acceder a los sistemas de ficheros paralelos a través de la interfaz ofrecida por MPI-IO.

Los patrones de acceso de las aplicaciones científicas paralelas consisten en un gran número de accesos pequeños y no contiguos. Para tamaños de acceso pequeños, el rendimiento viene limitado por la latencia de las transferencias de red y disco. Además, las aplicaciones científicas llevan a cabo accesos con una alta localidad espacial entre los distintos procesos en los nodos de E/S. Adicionalmente, las aplicaciones científicas presentan típicamente un comportamiento repetitivo. Cuando los patrones de acceso de E/S son repetitivos, técnicas como escritura demorada y lectura adelantada pueden enmascarar de forma eficiente las latencias de los accesos de E/S. Estas características han motivado a muchos investigadores en proponer optimizaciones de E/S tanto a nivel de biblioteca como a nivel del sistema de ficheros. Sin embargo, actualmente estas optimizaciones no se integran siempre a través de las distintas capas del sistema.

El objetivo principal de esta tesis es proponer una nueva arquitectura genérica de E/S paralela para clusters y supercomputadores. Nuestra solución está basada en una arquitectura de caches en varias capas, una técnica de E/S colectiva y estrategias de acceso asíncronas que ocultan la latencia de transferencia de datos entre las distintas capas de caches.

Nuestro diseño está dirigido a arquitecturas paralelas escalables con miles de nodos de cómputo. Además de actuar como middleware para los sistemas de ficheros paralelos existentes, nuestra arquitectura debe proporcionar virtualización on-line de los recursos de almacenamiento. Otro de los objeticos marcados para esta tesis es la factorización de las funcionalidades comunes en clusters y supercomputadores, en módulos genéricos que faciliten el despliegue de las aplicaciones científicas a través de estas plataformas.

Se han desplegado distintos prototipos de nuestras soluciones tanto en clusters como en supercomputadores. Las evaluaciones de rendimiento demuestran que gracias a la combicación de las estratégias colectivas de E/S y del solapamiento de computación, comunicación y E/S, se puede obtener una sustancial mejora del rendimiento en los patrones de acceso anteriormente descritos, muy comunes en las aplicaciones paralelas de caracter científico.

Contents

List of Figures vii

List of Tables ix

1 Introduction 1
- 1.1 Motivation . . . 1
- 1.2 Objectives . . . 3
- 1.3 Structure and Contents . . . 3

2 State of the art 5
- 2.1 Parallel I/O architectures . . . 5
  - 2.1.1 Clusters . . . 6
  - 2.1.2 Supercomputers . . . 7
- 2.2 Parallel file systems . . . 9
- 2.3 Message Passing Interface (MPI) . . . 15
  - 2.3.1 MPI data types . . . 15
  - 2.3.2 MPI-IO file model . . . 15
  - 2.3.3 ROMIO architecture . . . 17
- 2.4 Large-scale scientific data formats . . . 17
- 2.5 Parallel I/O optimizations . . . 18
  - 2.5.1 Non-contiguous I/O . . . 18
  - 2.5.2 Collective I/O . . . 19
- 2.6 File access latency hiding optimizations . . . 21
  - 2.6.1 Write-back . . . 22
  - 2.6.2 Prefetching . . . 22
- 2.7 Characterization of I/O accesses in HPC environments . . . 23
- 2.8 Summary . . . 24

3 Generic design of the parallel I/O architecture 25
- 3.1 Generic parallel I/O architecture . . . 26
- 3.2 Generic data staging . . . 28

3.3 Generic client-side file cache management . . . 29
3.3.1 File cache module . . . 29
3.3.2 File view management module . . . 30
3.3.3 Data staging modules . . . 34
3.3.4 File access semantics and consistency . . . 38
3.4 Summary . . . 39

4 Parallel I/O architecture for clusters **41**
4.1 Architecture overview . . . 41
4.2 File-system independent solution . . . 43
4.2.1 Architecture overview . . . 43
4.2.2 Design and implementation . . . 44
4.3 File-system specific solution . . . 51
4.3.1 Preliminaries . . . 51
4.3.2 Architecture overview . . . 52
4.3.3 Design and implementation . . . 53
4.4 Summary . . . 55

5 Parallel I/O architecture for supercomputers **57**
5.1 Preliminaries . . . 58
5.1.1 Network topology . . . 58
5.1.2 Operating system architecture . . . 59
5.2 Architecture description . . . 60
5.3 Data staging . . . 62
5.3.1 I/O node-side write-back . . . 63
5.3.2 I/O node-side prefetching . . . 64
5.4 Discussion . . . 66
5.5 Summary . . . 66

6 Evaluation **69**
6.1 Benchmarking setup . . . 69
6.2 Evaluation on clusters . . . 71
6.2.1 View-based I/O . . . 71
6.2.2 AHPIOS . . . 78
6.2.3 GPFS-based I/O . . . 86
6.3 Evaluation on supercomputers . . . 93
6.3.1 Blue Gene/L . . . 93
6.3.2 Blue Gene/P . . . 96
6.4 Summary . . . 113

7 Final remarks and conclusions **115**
7.1 Contributions 116
7.2 Thesis results 117
7.3 Future directions 119
7.3.1 Supercomputers 119
7.3.2 Parallel I/O architecture for multi-core systems 119
7.3.3 Cloud computing 120
7.3.4 High-performance POSIX interface 120
7.3.5 Parallel I/O system for Windows platforms 121

Bibliography **121**

List of Figures

2.1 Possible architectures for an private parallel I/O system. . . . 6
2.2 Clusters I/O architecture overview. . . . 7
2.3 Blue Gene architecture overview. . . . 8
2.4 Typical GPFS installations. . . . 10
2.5 Systems in a Lustre cluster. . . . 12
2.6 PVFS architecture. . . . 13
2.7 Expand parallel file system. . . . 14
2.8 pNFS file-based architecture. . . . 15
2.9 MPI file model. . . . 16
2.10 MPI views. . . . 16
2.11 Parallel I/O software architecture. . . . 17
2.12 The disk-directed collective. . . . 20
2.13 Two phase I/O write example. . . . 21

3.1 Generic parallel I/O architecture overview. . . . 26
3.2 Mapping view and lists of offset/lenght of two-phase I/O. . . . 31
3.3 View-based I/O data access operations steps. . . . 32
3.4 Comparison of two-phase I/O and view-based I/O. . . . 33

4.1 General parallel I/O architecture for clusters. . . . 42
4.2 AHPIOS software architecture with one application and one AHPIOS partition. . . . 44
4.3 AHPIOS partitions accessed by two MPI applications. . . . 46
4.4 Comparison of data flow in 2P I/O, client- and server-directed I/O. . . . 48
4.5 Comparison of two-phase I/O, view-based I/O, and GPFS-based I/O. . . . 53

5.1 File I/O forwarding for IBM solution. . . . 59
5.2 File I/O forwarding in ZeptoOS. . . . 60
5.3 Blue Gene cache architecture organized on six tiers. . . . 61
5.4 File mapping example in Blue Gene. . . . 63

6.1 BTIO data decomposition for 9 processes. . . . 70

6.2 View-based I/O. BTIO performance for class B. 72
6.3 View-based I/O. BTIO performance for class C. 73
6.4 View-based I/O. Breakdown per process for BTIO class B. 75
6.5 View-based I/O. MPI-TILE-IO performance. 76
6.6 View-based I/O. Coll perf performance. 77
6.7 AHPIOS. Scalability. 80
6.8 AHPIOS. BTIO class B measurements. 81
6.9 AHPIOS. BTIO class C measurements. 82
6.10 AHPIOS. MPI Tile I/O throughput. 84
6.11 GPFS-based I/O. gpfsPerf performance for strided pattern access. 87
6.12 GPFS-based I/O. FLASH I/O performance. 88
6.13 GPFS-based I/O. BTIO class B file write performance. 89
6.14 GPFS-based I/O. gpfsPerf file read performance for a strided access pattern. . . . 90
6.15 GPFS-based I/O. SimParIO file read performance for strided access pattern. . . . 91
6.16 GPFS-based I/O. gpfsPerf file read performance for a strided access pattern. . . . 92
6.17 GPFS-based I/O. BTIO class B file read performance. 92
6.18 BG/L. Performance of contiguous access for a fixed record size of 128KBytes. . . 94
6.19 BG/L. Performance of contiguous access for a fixed record size of 1MByte. 95
6.20 BG/L. Performance of contiguous access for 64 processes. 95
6.21 BG/L. Performance of contiguous access for 512 processes. 96
6.22 BG/L. BTIO class C times for 64 processes. 97
6.23 BG/L. BTIO class C times for 256 processes. 97
6.24 BG/L. Times for all steps of BT-IO class C. 98
6.25 BG/P. Torus network throughput for different message sizes. 100
6.26 BG/P. Tree network throughput for different message sizes. 100
6.27 BG/P. PVFS file write and read throughput on an single I/O node. 100
6.28 BG/P. Effect of number of aggregators on the aggregate file write throughput. . . 101
6.29 BG/P. Effect of number of aggregators on the aggregate file write throughput. . . 102
6.30 BG/P. Effect of number of aggregators on the aggregate file write throughput. . . 102
6.31 BG/P. Effect of number of aggregators on the aggregate file read throughput. . . 103
6.32 BG/P. Effect of number of compute nodes on the aggregate write throughput. . . 104
6.33 BG/P. Effect of number of compute nodes on the aggregate read throughput. . . 104
6.34 BG/P. Effect of file cache sizes on the aggregate file write throughput. 105
6.35 BG/P. File write performance for 64 processes. 107
6.36 BG/P. File write performance for 256 processes. 108
6.37 BG/P. Aggregate write throughput and file close time for 64 nodes. 109
6.38 BG/P. Effect of prefetching window on the aggregate file read throughput. 110
6.39 BG/P. Histogram of the 40 phases of file read for the 16 blocks read-ahead. . . . 111
6.40 BG/P. BTIO class B file write times for 64 and 256 processes 112

6.41 BG/P. BTIO class B file read times for 64 processes. 112
6.42 BG/P. Histograms of the 40 file read operations of BTIO class B times 113

List of Tables

2.1 Description of GPFS hints structures. 11

3.1 Description of variables and functions used by algorithms. 35

4.1 Comparison of three collective I/O methods. 50

5.1 Comparison of the BG/L and BG/P systems. 58

6.1 Granularity of BTIO file accesses. 70
6.2 The overhead of lists of length-offset pairs (in KBytes) of BTIO Class B. 74
6.3 The count of MPI point to point and collective communication. 78
6.4 Parameters of MPI Tile I/O benchmark. 83
6.5 The count of MPI point-to-point operations. 85
6.6 The count of MPI collective operations. 85
6.7 Experiment parameters on BG/P. 99

Chapter 1

Introduction

The recent advances in storage technologies and high performance interconnects have made possible in the last years to build more and more potent storage systems that serve thousands of nodes. In addition, the increasingly hybrid and hierarchical design of memory [MEFT96], with multi-level caches that can be exclusive to or shared between the processor cores, as well as NUMA-style memory access will pose further roadblocks to achieving high performance on modern architectures.

In 1987, Gray and Putzolo published their well-know five-minute rule [GP87, GG97] for trading-off memory and I/O capacity. Their calculation compares the cost of holding a record permanently in memory with the cost to perform disk I/O each time the record is accessed, using appropriate fractions of prices for RAM chips and for disk drives. The name of their rule refers to the break-even interval between accesses. If a record is accessed more often, it should be kept in memory; otherwise, it should remain on disk and read when needed.

1.1 Motivation

In the last years the computing power of high-performance systems has continued to increase at an exponential rate, making even more challenging the access to large data sets. The ever-increasing gap between I/O subsystems and processor speeds has driven researchers to look for scalable I/O solutions, including parallel file systems and libraries. A typical *parallel file system* stripes the file data and metadata over several independent disks managed by I/O nodes in order to allow parallel file access from several compute nodes. Examples of popular file systems include GPFS [SH02], PVFS [LR99], and Lustre [Inc02]. These parallel file systems manage the storage of several clusters and supercomputers from the top 500 list. *Parallel I/O libraries* include MPI-IO [Mes97], Hierarchical Data Format (HDF) [HDF], and parallel NetCDF [LLC+03].

The parallel I/O solutions proposed so far in literature address either clusters of computers or supercomputers. Given the proprietary nature of many supercomputers, the majority of works has concentrated on clusters of computers. Only a limited number of papers have proposed novel

solutions for scalable parallel I/O systems of large supercomputers. Nevertheless, supercomputers systems have a complex architecture consisting of several networks, several tiers (computing, I/O, storage) and, consequently, a potential deep cache hierarchy. This type of architecture provides a rich set of opportunities for parallel I/O optimizations. Existing approaches are not suitable for a multi-tier architecture, in the sense that they do not provide an integrated coherent cache hierarchy. Additionally, most file operations are synchronous, i.e. there is a low degree of overlapping between operations at different layers.

The I/O access patterns of scientific parallel applications often consist of accesses to a large number of small, non-contiguous pieces of data. Furthermore, many current data access libraries such as HDF5 and netCDF rely heavily on small data accesses to store individual data elements in a common large file [HDF, LLC+03]. For small file accesses the performance is dominated by the latency of network transfers and disks. Additionally, parallel scientific applications lead to interleaved file access patterns with high interprocess spatial locality at the I/O nodes [NKP+96, SR98]. These characteristics of the access patterns motivated several researchers to propose parallel I/O optimizations both at library and file system levels. However, these optimizations are not always integrated across different layers. In this thesis, we propose ourselves to demonstrate that a tighter integration results in a significant performance improvements, scalability, and better resource usage.

Besides the performance, the productivity is taken into consideration, which includes the costs of programming, debugging, testing, optimization and administration. In case of parallel I/O, it becomes more difficult to obtain optimal performance from the underlying parallel file system, given different I/O requirements. The default file system configuration cannot always provide an optimal throughput for different data intensive applications. File system reconfiguration may be a solution, but an expensive one, that would inevitably involve the administrative overhead, data relocation, and system down time. Additionally, the design complexity of a distributed parallel file system such as GPFS, makes it difficult to address the requirements from different I/O patterns at file system level. In other words, implementing the solutions for these problems in the file systems would come at the cost of additional complexity. However, the solutions addressing specific I/O requirements can be done at a higher level, closer to the applications.

The work proposed in this thesis starts from the following premises:

- The parallel scientific applications produce and consume a huge increasing amount of data that has to be transferred to and from persistent storage.
- The current parallel computer architectures reached a Pflops (a thousand trillion floating point operations per second) performance and target the Eflops milestone. This scale involves a deep hierarchy of memory and storage through which data is to be transferred. To the best of our knowledge there are few solutions that address this problem in an efficient manner.
- The complexity of a deep hierarchy offers a rich set of opportunities for optimization in order to improve the performance of file accesses based on the access patterns of scientific applications.
- Despite of the large spectrum of parallel computer architectures, we consider that there are common denominators of parallel I/O optimizations, which can be applied independently of architecture-specific hardware and software configurations.

1.2 Objectives

The major objective of this thesis is to propose a generic parallel I/O architecture for both clusters and supercomputers. Additionally, the thesis targets the following objectives:

- **High-performance parallel I/O**. The thesis targets to reduce the file access latency perceived by the data-intensive parallel scientific applications by multi-layer asynchronous data transfers. In order to accomplish this objective, our techniques leverage the multi-core architectures by overlapping computation with communication and I/O in parallel threads.
- **Scalability**. Our design and implementations is aimed at large-scale parallel architectures with thousands of compute nodes.
- **Dynamic virtualization**. Besides acting as middleware for existing parallel file systems, our architecture must provide on-line virtualization of storage resources. This approach targets the goal of offering a simple ad-hoc parallel I/O system for the machines not having a parallel file system installed.
- **Portability**. The implementation of our architecture must be portable using standards such as MPI and POSIX [hs95]. Additionally, the architecture is transparent to the applications, which do not have to be modified.
- **Cross platform software factorization**. One of the objectives of this thesis is to factor out the common parallel I/O functionality from clusters and supercomputers in generic modules in order to facilitate the porting of scientific applications across these platforms.

1.3 Structure and Contents

The remainder of this document is structured in the following way.

- Chapter 2 *State of the art* contains the state of the art.
- Chapter 3 *Generic design and implementation of the I/O architecture* presents the generic design and implementation our generic parallel I/O architecture. Additionally, this chapter is dedicated to present the common denominator of both architectures.
- Chapter 4 *Architecture for clusters* describes how the generic parallel I/O architecture can be applied for clusters of off-the-shelf components. We present two uses of the proposed parallel I/O architecture for independent and dependent I/O systems.
- Chapter 5 *Architecture for supercomputers* shows how the generic parallel I/O architecture from the previous section can be applied for supercomputers. In particular, we present our architecture evolved from the Blue Gene/L to Blue Gene/P.
- Chapter 6 *Evaluation* reports performance results for both cluster and supercomputer systems.
- Chapter 7 *Final Remarks and Conclusions* contains a summary of this thesis, publications, and future plans.

Chapter 2

State of the art

This chapter presents the state of the art related to this dissertation and the background concepts necessary for understanding the solution. The material is organized in four subsections: parallel I/O architectures, parallel file systems, MPI library and parallel I/O optimizations, and a brief characterization of I/O in HPC environments.

2.1 Parallel I/O architectures

Parallel computers are known to have a number of different architectures [Buy99]. The Massively Parallel Processor (MPP) is a large parallel system. Normally, it consists of several hundred processing elements (nodes). The nodes are connected via high-speed network and each of them runs a separate copy of an operating system. MPP implements a *shared-nothing* architecture. Often, each node consists only of main memory and one or more processors. Additional components, like I/O devices, could be added. In contrast, the Symmetric Multiprocessor (SMP) system has a *shared-everything* architecture. Each node can also use the resources of the other nodes (memory, I/O devices). A single operating system runs on all the nodes. Any combination of MPPs, SMPs or plain computers could be added to a distributed system.

We describe the most commonly deployed parallel I/O architectures for both clusters and supercomputers.

An internal parallel I/O subsystem has advantages over external servers. Compared to a mass-storage system, the subsystem can handle requests with lower latency. Communication between compute nodes and I/O nodes occurs though a reliable, low-latency, message-passing protocol, or through shared memory, rather than through the slower network protocol used by LAN-connected file servers (SAN). Also, the subsystem can more effectively store data for sharing and reuse. Compared to a group of small file servers connected by a LAN, subsystem still offers lower latency and greater bandwidth.

In the simplest architecture disks are attached to compute nodes (see Figure 2.1a). Using local disks for persistent files would make it difficult to collocate applications and required data.

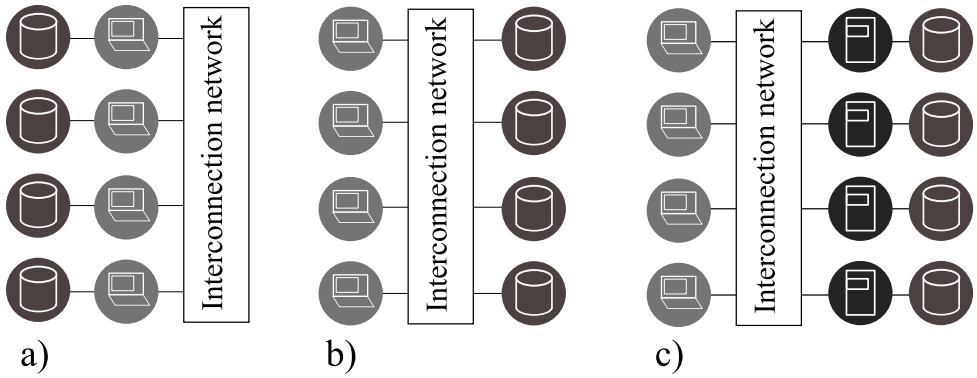

Figure 2.1: *Possible architectures for an internal parallel I/O system: (a) attaching disks to compute nodes (CN), (b) attaching disks to the network, and (c) separate I/O nodes with own processor, memory, and disks.*

For example, if a certain file is stored on the disk attached to a specific node, an application using that file should execute on that node. If that node is running another application, executing the new application on that node would cause load imbalance. Executing it elsewhere would lead to internode communication for the I/O. This communication could interfere with the other application's execution.

If the disks are attached to the network, rather than directly to specific nodes (see Figure 2.1b), data cannot be collocated with a node that requires it. All compute nodes can access each disk, so each node should be able to access any data, with equal performance. Consider two applications writing two different files that reside on the same disk. The system software on the nodes running the applications must coordinate the block allocation on the disk, so that space is allocated properly. This requires some shared data structures with locks, and an access protocol to update them. In addition, there is no obvious place to buffer data of files shared by more than one process.

Most vendors select an internal I/O subsystem architecture that uses separate I/O nodes, complete with processor, memory, and disks (see Figure 2.1c). Each parallel file is distributed across the I/O nodes. Each I/O node stores portions of parallel files, and handles all block allocation and buffering of those portions. Simultaneous data movement between the I/O and compute nodes of different applications is possible, without any direct coordination among the compute nodes.

A parallel I/O subsystem based on multiple I/O nodes allows data transfer in parallel between compute nodes and I/O nodes. This architecture presents the following advantages: first, it can efficiently handle the small, fragmented requests produced by parallel programs. Second, by adding I/O nodes or disks, it can scale in bandwidth and capacity to keep up with increases in the number and speed of compute nodes. Finally, the parallel subsystem provides load distribution by scattering I/O operations across multiple nodes. It also can provide reliability when a system element is down or being replaced.

2.1.1 Clusters

High performance I/O for cluster architectures has been a subject for a lot of research [BMV03, Aba03, OT04]. While several big clusters have adopted low cost-per-node distributed disk (DD)

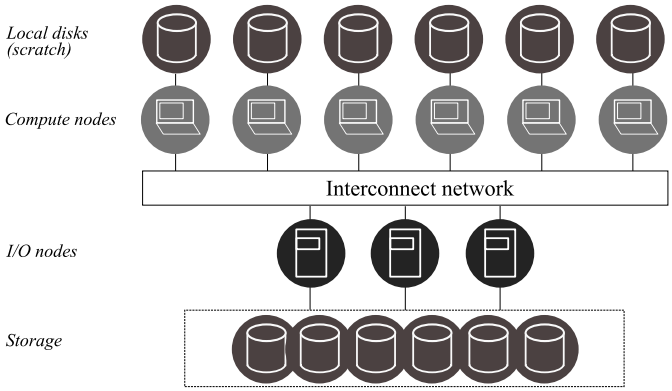

Figure 2.2: *Clusters I/O architecture overview.*

architectures where I/O nodes have inexpensive internal disks, small-to-medium sized clusters, used in scientific research and in business data infrastructures to support large data bases, have been favoring the shared disk (SD) approach.

In DD architectures compute nodes perform data movement to and from I/O nodes either across inexpensive Ethernet interconnects, or via more expensive specialized ones such as Myrinet [Myr], SCI, or Infiniband [Ass98]. In SD architectures, both nodes and storage devices are attached to a storage area network (SAN), an infrastructure that commonly uses Fiber Channel. The cost per node for both types of infrastructures is similar.

This diversity among architectures has obviously spurred different file system approaches: distributed disk architectures are usually handled with a distributed file system (DFS), while shared disk architectures resort to cluster file systems (CFS). Some file systems are specifically designed to cater for HPC needs such as Parallel Virtual File System (PVFS) and General Parallel File System (GPFS).

2.1.2 Supercomputers

Today's Massively Parallel Processing (MPP) platforms reach with 100s of Tflops to Pflops [ACI+09]. In order to meet the needs of data-intensive scientific applications, MPP such as Blue Gene [SHea06, Mea06, ABB+08, DHJ+04] and Cray XT [VAD+06, AKB+07, Rot07], require a scalable I/O subsystem, both in hardware and software.

Blue Gene architecture. Blue Gene is a massively parallel computer developed at the IBM Thomas J. Watson Research Center. Blue Gene/P (BG/P) is the second generation of Blue Gene solutions from IBM, following Blue Gene/L (BG/L). IBM designed these systems as completely customized architectures for high performance computing that balance performance with the requirements for low power, scalability, and high density. In early 2008, BG/L systems lead the Top 500 list, holding 21 slots, with BG/P holding five slots. Ten of the top 50 systems on the list were from the BlueGene family. Perhaps more impressive is the fact that BG/P and BG/L own the top 26 spots on the Green500 list, which ranks systems by their power efficiency.

Figure 2.3 shows a high-level view of Blue Gene systems. Compute nodes are specialized to run user applications and the I/O nodes only perform system functions related to I/O. Com-

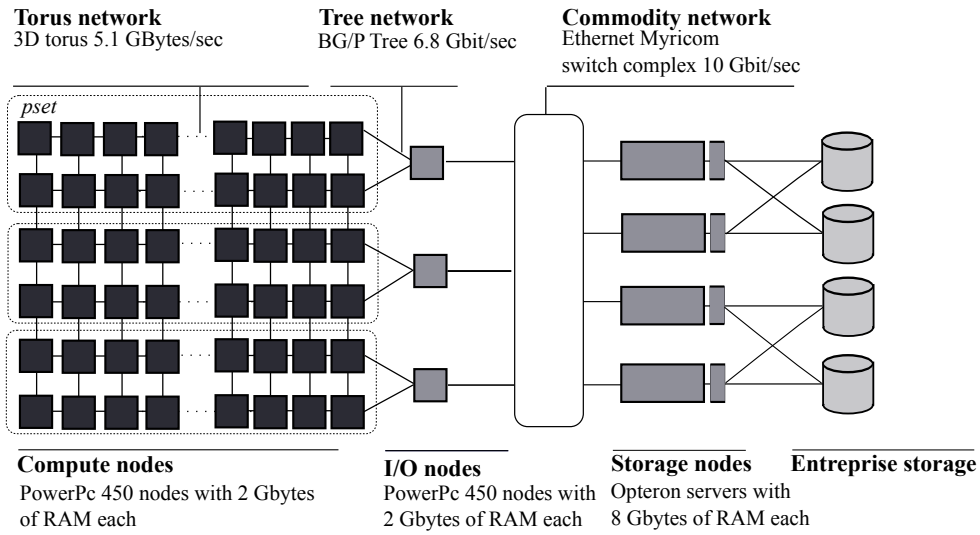

Figure 2.3: *Blue Gene architecture overview. Applications run on compute nodes. Compute nodes are grouped into processing sets, or "psets". Each pset has an associated I/O node, which performs I/O operations on behalf of the compute nodes from the "psets". The compute and I/O nodes are controlled by service nodes. The file system components run on dedicated file servers connected to storage nodes.*

pute nodes are grouped into processing sets, or "pset". Each "pset" has an associated I/O node, which performs I/O operations on behalf of the compute nodes from the "pset". The file system components run on dedicated file servers connected to storage nodes.

Blue Gene nodes are interconnected by five different networks: 3D torus, collective, global barrier, Ethernet and control. The 3D torus is typically used for point-to-point communication between compute nodes. The collective network has a tree topology and serves collective communication operations and I/O traffic. The global barrier network offers an efficient barrier implementation. The Ethernet network interconnects I/O nodes and file servers. The service nodes control the whole machine through the control network.

A Blue Gene system can be divided in partitions. At a given time each partition executes only one job. For example, in Blue Gene/P, the compute nodes of a partition may run in three modes: symmetric multi-processing (SMP), dual, and virtual modes. In SMP mode, the compute node executes a single MPI task per node with a maximum of four threads within the task. Dual mode is a new mode for the BG/P systems. In this mode, each compute node executes two MPI tasks per node with a maximum of four threads on the compute node (two per MPI task). Memory and cores are split evenly between the two tasks. In virtual mode, each of the cores in the processor has an MPI rank and runs a program process. There is no additional threading capability in virtual node mode.

Cray XT architecture. Cray XT is a parallel computing platform that features massive parallelism and high performance. An XT system consists of processing elements (PEs) connected in a three-dimensional mesh or torus topology. Each PE contains one or two AMD Opteron processors, memory, and a Cray proprietary router Application-Specific Integrated Circuit (ASIC) called SeaStar. Single-core and multi-core processors are supported.

Cray XT PEs are partitioned into compute PEs and service PEs. Compute PEs run applica-

tion processes, and use either a lightweight operating system kernel called Catamount or Cray's Compute Node Linux (CNL). Service PEs provides login and I/O services with a traditional Linux installation.

The I/O subsystem of Cray XT is provided through Lustre parallel file system (more details about parallel file systems are given in Section 2.2). Cray provides a proprietary MPI-IO implementation over Lustre on Cray XT [YVC07a]. Larkin and Fahey [FLA08] analyze the performance of Lustre on the Cray XT3/XT4, and provide some guidelines to maximize I/O performance on this platform.

2.2 Parallel file systems

A parallel file system is a basic component of any scalable parallel I/O solution. According to [TU99], a parallel file system is a file system that avoids the I/O bottleneck joining in a logical way independent storage devices and I/O nodes by means of a high performance storage system. The bandwidth is increased because of the independent addressing of the disks (access to data of different files in a concurrent way) and data de-clustering (access to data of the same file in parallel).

In this section we summarize the main characteristics of some historical, commercial, and research parallel file systems.

Vesta. The Vesta [CF96] parallel file system provides parallel access from compute nodes to files distributed across I/O nodes in a massively parallel computer. Vesta is intended to solve the I/O problems of massively parallel computers executing numerically intensive scientific applications. Vesta has three interesting characteristics: First, it provides a user defined parallel view of file data, and allows user defined partitioning and repartitioning of files without moving data among I/O nodes. The parallel file access semantics of Vesta directly support the operations required by parallel language I/O libraries. Second, Vesta is scalable to a very large number (many hundreds) of I/O and compute nodes and does not contain any sequential bottlenecks in the data-access path. Third, it provides user-directed checkpointing of files with continuing program execution with very little processing overhead.

Galley. Files in the Galley Parallel File System [NK97] are composed of one or more subfiles. Each subfile resides on a single disk and contains one or more forks that are contiguous segments stored on the respective disk. The underlying parallel structure of the file is hidden from the application. Galley offers a particular interface that allows simple strided, nested-strided and nested- batched operations. No file views are possible.

PPFS. The Portable Parallel File System (PPFS) [HER+95] allows applications to control caching, pre-fetching, data distribution and file sharing policies. The files are divided into variable size records, called segments. Each segment is managed by a single I/O server. PPFS is implemented as a user level library portable across several parallel file systems.

ParFiSys. ParFiSys is a coherent parallel file system developed at the UPM to provide I/O services to the GPMIMD machine, an MPP built within the ESPRIT project. The main goals of

ParFiSys [CSM+96, PCG+97] are to provide I/O services to scientific applications requiring high I/O bandwidth, to minimize application porting effort, and to exploit the parallelism inherent to generic message-passing multicomputers, including processing nodes (PN) and I/O nodes (ION). ParFiSys has been used to explore a variety of data distributions, distributed caching and prefetching strategies: disk data striping factors, file and disk block prefetch policies, caching policies, cache coherence models, and I/O patterns.

GPFS. The General Parallel File System (GPFS) from IBM grew out of the Tiger Shark multimedia file system [IBM98, SH02, AR, AHP01] and has been widely used on the AIX platform.

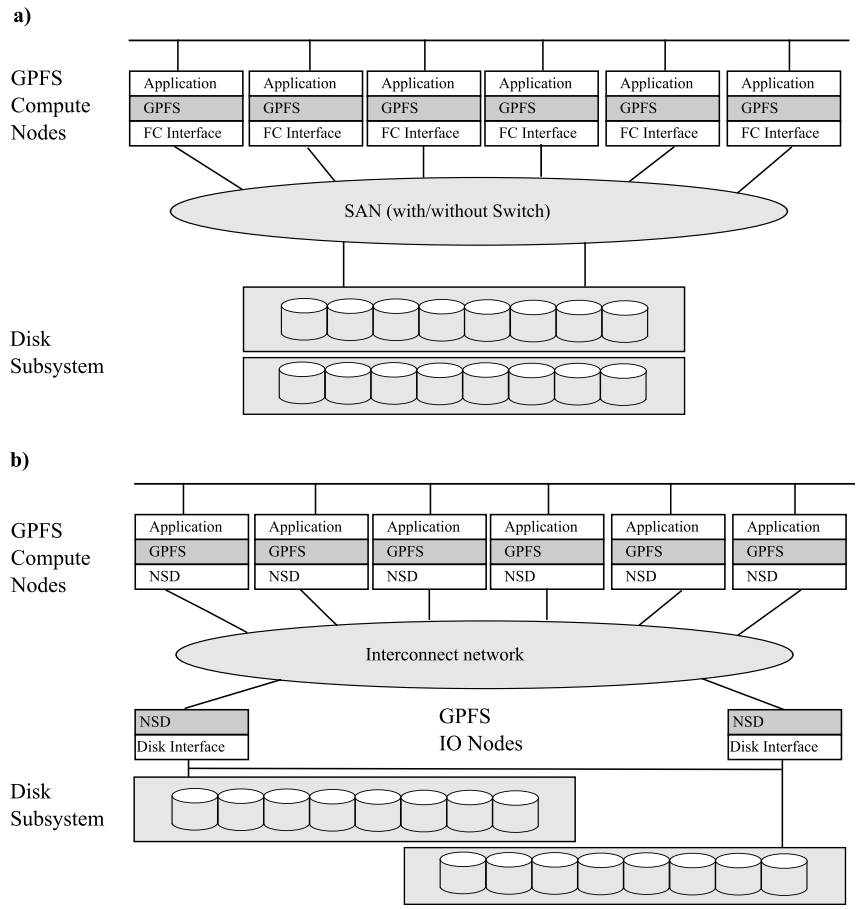

Figure 2.4: *Typical GPFS installations: a) SAN attached GPFS cluster b) NSD based GPFS cluster.*

GPFS is a parallel file system for clusters or supercomputers. Its architecture is based on shared disks, mounted by all client nodes. The data and metadata reside on shared disks and might be cached in clients's caches. In order to guarantee data coherency, GPFS relies on a distributed locking manager. Locks are acquired and kept by clients while caching data. The granularity of locking in GPFS is at the byte-range level, consequently, writes to non-overlapping data blocks of the same file can proceed concurrently.

Two typical installations of GPFS are shown in Figure 2.4 [IBM98]. In the simplest setup, the shared disks are connected to all nodes through a Storage Area Network (SAN) such as Fiber Channel, as depicted in the top side of Figure 2.4. The nodes are interconnected through a Local

Area Network (LAN) solution or low-latency high-throughput interconnects such as Infiniband, Myrinet or IBM pSeries High Performance Switch (HPS). The data transferred over the SAN network, while the GPFS control information over the LAN.

A second setup is used in the systems that do not have a SAN installation. As shown in the bottom side of Figure 2.4, in this case a Network Shared Disk (NSD) is exported to all the nodes through one I/O node (several I/O nodes can also be used for availability). The accesses to the I/O servers are transparent to the applications issuing file system calls. Both data and control information are transferred in this setup over a high-performance interconnect.

GPFS is highly optimized for large-chunk I/O operations with regular access patterns (contiguous or regularly strided). However, its performance for small-chunk, non-contiguous I/O operations with irregular access patterns (e.g. non-constant strides) is not sufficiently addressed. This kind of access can cause a high access contention, translated especially in a high locking overhead.

GPFS addresses this issue by a technique called data-shipping [SH02, PTH+01], which can be activated/deactivated through a hint (a configurable parameter) of the GPFS library. This technique disables client-side caching and binds each GPFS file block to a single I/O agent, which will be responsible for all accesses to this block. For write operations, each task sends the data to be written to the responsible I/O agents. I/O agents in turn issue the write calls to the end storage system. For reads, the I/O agents read the file blocks, and ship only the requested read data to the appropriate tasks. This approach is similar to the two-phase I/O, described in Section 2.5.2. Data-shipping is more efficient than the default locking approach, when fine-grained sharing is present, because the granularity of GPFS cache consistency is an entire file block, and accesses to the same block are serialized by the locking manager.

The three relevant GPFS hints are described in Table 2.1, which we have used in implementations presented in this thesis.

Table 2.1: *Description of GPFS hints structures.*

GPFS hint	Description
gpfsDataShipStart	Initiates data shipping mode for a file.
gpfsDataShipStop	Finalizes data shipping mode for a file.
gpfsMultipleAccessRange	Defines file blocks used for prefetching and write-behind.
gpfsClearFileCache	Invalidates the local buffer cache.

Lustre. The Lustre [Inc02, DL08] project aims at providing a file system for clusters of tens of thousands of nodes with petabytes of storage capacity. It is designed, developed and maintained by Oracle.

Lustre is a POSIX compliant, object-based file system composed of three components: Metadata Servers (MDS), Object Storage Servers (OSSs), and Object Storage Targets (OSTs), as shown in Figure 2.5. A Lustre file system has one or more Object Storage Servers (OSSs), which handle interfacing between the client and the physical storage. Each OSS serves one or more Object Storage Targets (OSTs), where the file objects are stored to disk. The file system names-

pace is served by a Metadata Server (MDS). As the name implies, the MDS is a database that holds the file metadata for the entire file system. Whenever a metadata operation occurs, such as an open or file creation, the client must poll the MDS. At this time, a Lustre file system is limited to one MDS, which can result in a parallel I/O performance bottleneck at large scale [FLA08, VAD+06].

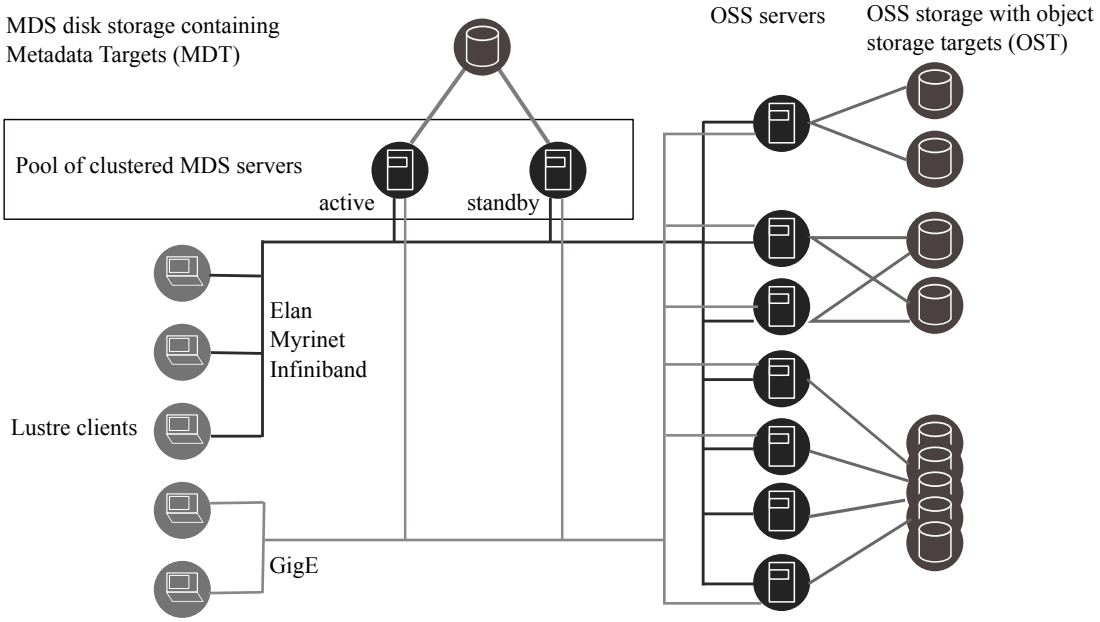

Figure 2.5: *Systems in a Lustre cluster. In a Lustre file system, storage is only attached to the server nodes, not to the client nodes.*

In [PNF07], the authors show how offloading computation at the Lustre storage nodes (active storage) may reduce communication and boost performance. Using Lustre file joining (merging two files into one) for improving collective I/O is presented in [Wei07].

PVFS. One of the most popular parallel file systems is PVFS (Parallel Virtual File System) [LR99]. PVFS is an open source parallel file system that targets the efficient access to large data sets. PVFS consists of several server processes and several client nodes accessing the file system through a proprietary library or through a Linux kernel module implementing a mountable VFS interface. Each server may be both a data and or a metadata manager. Efficient non-contiguous I/O may be performed through the list I/O interface (in which we will go into more detail in Subsection 2.5.1). Figure 2.6 shows a typical PVFS configuration for very large-scale systems.

PVFS supports a set of feature-rich interfaces [CONP07]. PVFS can be used with multiple APIs: a native API, the UNIX/POSIX API, and MPI-IO. The presence of multiple popular interfaces contributes to the wide success of PVFS in both industry and university settings. The system interface API provides functions for the direct manipulation of file system objects and hides internal details from the user. Invoking a request starts a dedicated state-machine processing the operation in small steps. State-machines break complex requests into several states each representing an atomic operation.

The Buffered Message Interface (BMI) provides a network independent interface. Clients communicate with the servers by using the request protocol, which defines the message layout

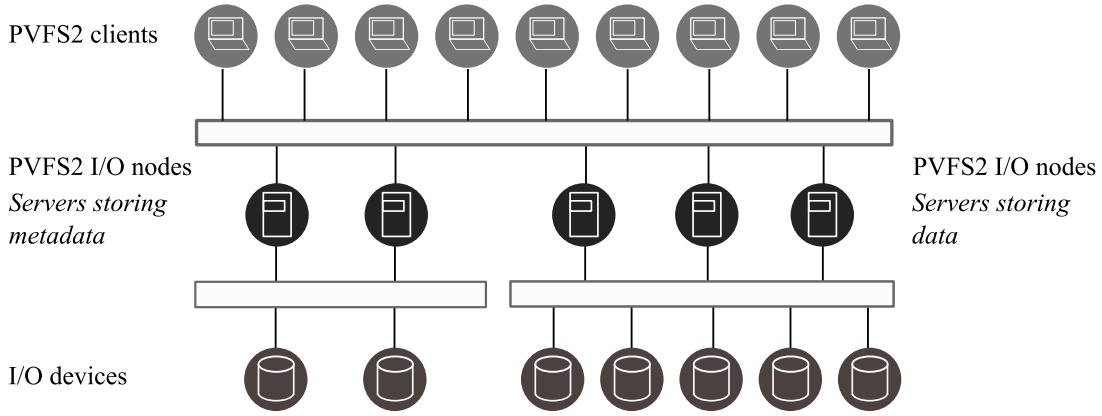

Figure 2.6: *PVFS architecture. Example of a very large-scale system.*

for every request. BMI can use different communication methods, currently TCP, Myricom GM and Infiniband. On the server side a main process decodes incoming requests and starts a new instance of the state-machine.

Expand. Expand (Expandable Parallel File System) [GCCC+03] is a parallel file system that combines multiple data servers (NFS servers) to create a distributed partition. Expand requires no changes to the data server and uses standard protocol operations to provide parallel access to the same file. Expand is also client-independent, because all operations are implemented using the data server protocols. Using this system, heterogeneous servers (Linux, Solaris, Windows 2000, etc.) can be joined into a parallel and distributed partition.

Expand (showed in Figure 2.7) combines several servers to provide a generic striped partition which allows to create several types of file systems by defining different types of partitions. A file in Expand consists of several subfiles, one for each server in the distributed partition. The decomposition of a file into subfiles is fully transparent to the users. Expand hides the file structure, offering to the clients a traditional view of the file. In order to exploit all the available storage resources, Expand provides several data allocation and load balancing algorithms that automatically search and select the available and proper servers for storing the data of a file.

Clusterfile. Clusterfile Parallel File System [IT01, IT03a, ISC+06] provides parallel file access on a cluster of computers. Existing parallel file systems offer little control over matching the I/O access patterns and file data layout.

A file is physically partitioned into subfiles stored at I/O servers and may be logically partitioned among several compute nodes by views. Views are implemented inside the file system, as opposed to MPI-IO views, which are implemented on top of file system. The subfiles or views partitions may be constructed either through MPI data types or through an equivalent, Clusterfile native data representation. A central metadata manager manages file inodes. Data and metadata management are separated, therefore, data does not travel through the metadata manager.

Clusterfile uses both disk directed I/O and two-phase I/O together with a global file cache. This tight integration, as well as the careful attention paid to extracting parallelism from other subsystems such as memory and global file cache, result in significant performance improvements.

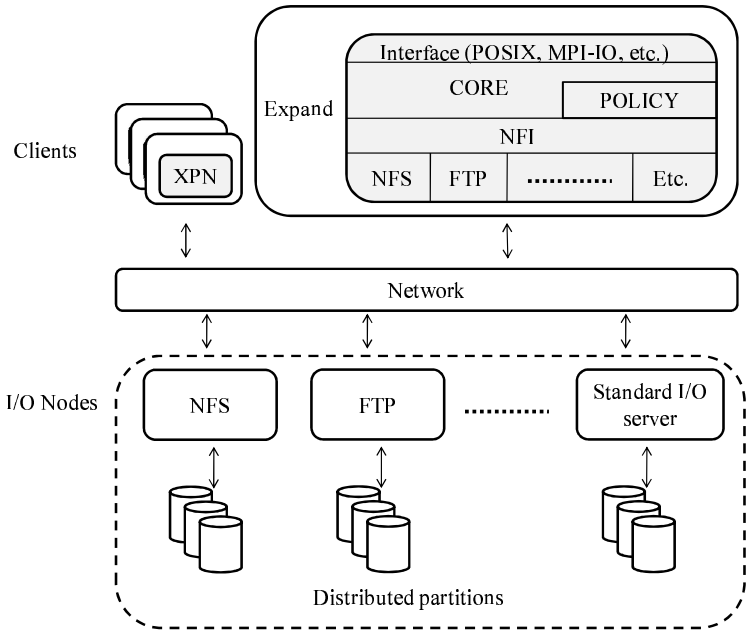

Figure 2.7: *Expand parallel file system. Expand combines multiple I/O servers to create a distributed partition where files are declustered.*

pNFS. pNFS [HH05] is an extension of the NFS protocol (in the version 4.1) and provides parallel access to storage systems. pNFS is a continuation of efforts on parallelizing the file access by striping files over multiple NFS servers [GCCC+03, KMM94, LD02]. pNFS is likely to become the de-facto standard of high performance parallel storage access and has already been adopted by storage leader companies such as Panasas and IBM. Panasas is migrating its PanFS parallel file system [htt08d] to pNFS. IBM is also implementing pNFS on top of GPFS. pNFS provides a storage system independent of operating system and allows client applications to fully utilize the throughput of a shared parallel file system.

A pNFS file-based system consists of pNFS data servers, clients and a metadata server, plus parallel file system (PFS) storage nodes, clients, and metadata servers (as shown in Figure 2.8). The three-tier design prevents direct storage access and creates overlapping and redundant storage and metadata protocols [HH07]. The two-tier design, which places pNFS servers and the exported parallel file system clients on storage nodes, suffers from these problems plus diminished single client bandwidth.

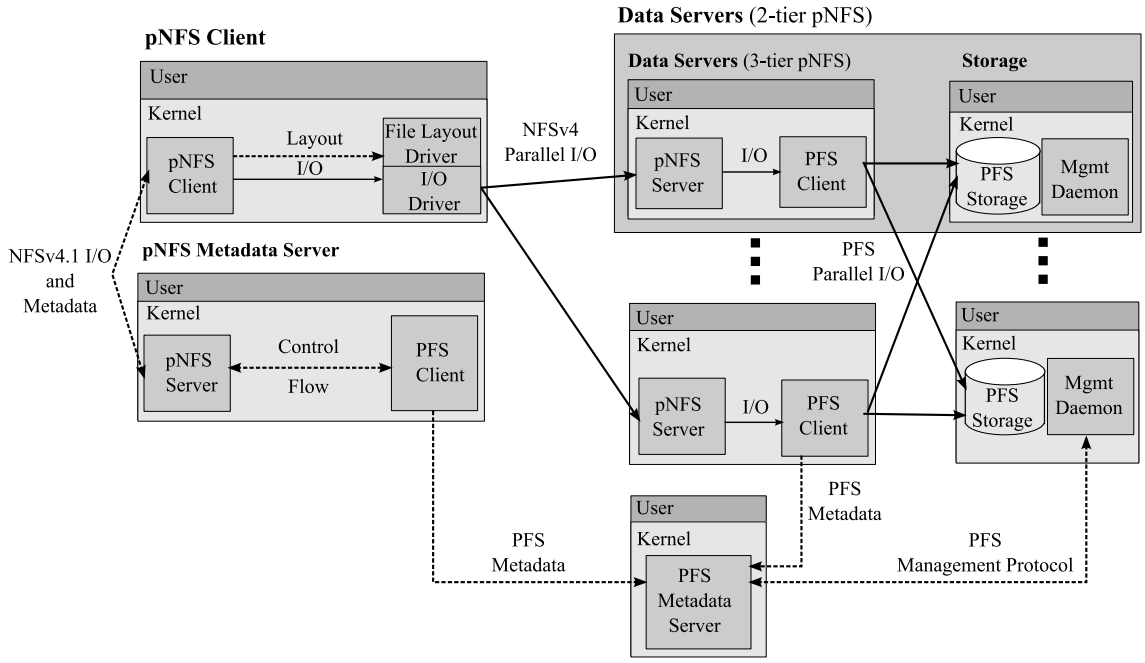

Figure 2.8: *pNFS file-based architecture. The pNFS client uses layout and I/O drivers to access storage nodes and follow underlying file system polices. The pNFS server uses PFS export operations to exchange pNFS information with the underlying file system.*

2.3 Message Passing Interface (MPI)

The large majority of large-scale scientific parallel applications are written in Message Passing Interface (MPI) [Mes95], which has become the de-facto standard for scalable distributed-memory machines. One part of the MPI standard is related to I/O and has among its main goals the portability and efficiency of file system accesses. All of the above mentioned parallel file systems may be accessed also through the MPI-IO interface.

This section introduces some basic concepts of MPI (data types, views, I/O operations, file model) and presents the ROMIO architecture.

2.3.1 MPI data types

MPI data types are access patterns of memory or file. They can express regular or irregular patterns, with or without gaps between the data. The *basic* data types are the same as in traditional programming languages like C. The *derived* data types are built from basic data types or recurrently from other derived data types. Vectors are examples of derived data types.

2.3.2 MPI-IO file model

An MPI file is an ordered collection of typed data items. A file is opened collectively by a group of processes represented by a communicator. All collective I/O calls on a file are collective over this group.

A file *displacement* is an absolute byte position relative to the beginning of a file. The

displacement defines the location where a view begins.

An *etype* (elementary datatype) is the unit of data access and positioning. It can be any MPI predefined or derived datatype. Data access is performed in etype units, reading or writing whole data items of type etype. Offsets are expressed as a count of etypes.

A *filetype* is the basis for partitioning a file among processes and defines a template or accessing the file. A filetype is either a single etype or a derived MPI datatype constructed from multiple instances of the same etype.

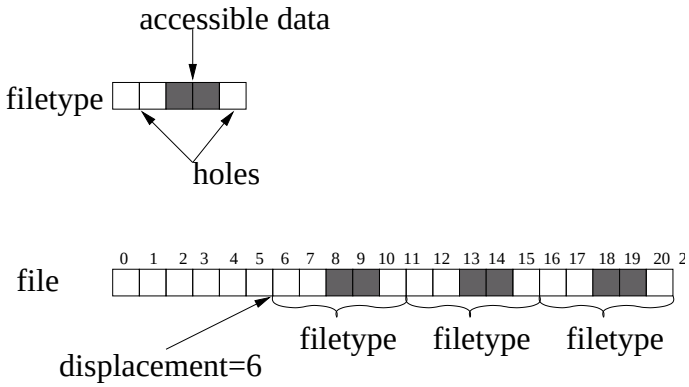

Figure 2.9: *MPI file model.*

A *view* [IT03b] defines the current set of data visible and accessible from an open file as an ordered set of etypes. Each process has its own view of the file, defined by three parameters: a displacement, an etype, and a filetype. The pattern described by a filetype is repeated, beginning at the displacement, to define the view. The view from Figure 2.9 starts at displacement 6, has etype of 1 byte and a filetype of extent 5, out of which only 2 bytes are accessible. The default view is a linear byte stream (displacement is zero, etype and filetype are equal to MPI_BYTE), i.e. the view maps one-to-one to the file. Views offer several advantages: non-contiguously stored data is "seen" as contiguously facilitating the programmer's task and allowing non-contiguous I/O optimizations. At same time a view is a hint about the future access pattern of the application, which can be used for optimizing the access.

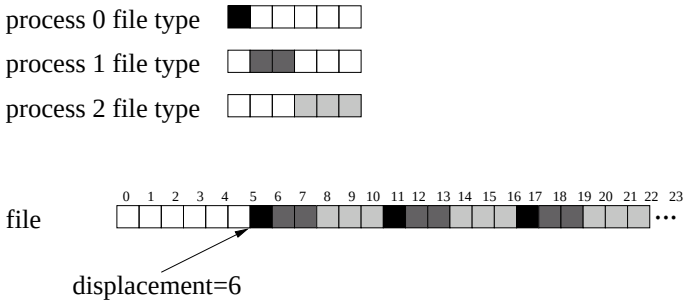

Figure 2.10: *MPI views.*

A group of processes can use complementary views in order to achieve a global data distribution such as a scatter/gather pattern (see Figure 2.10).

An offset is a view position, expressed as a count of etypes. Holes in the view filetype are

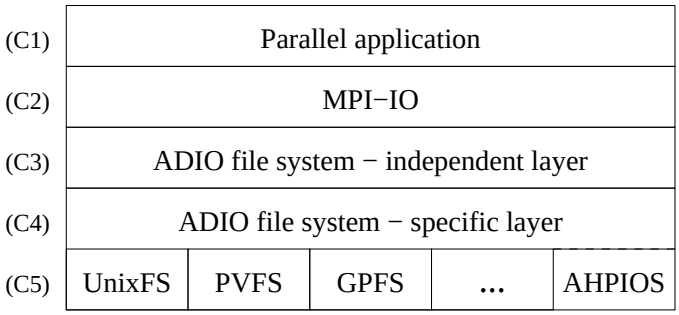

Figure 2.11: *Parallel I/O software architecture.*

skipped when calculating this position. Offset 0 is the location of the first etype visible in the view (after skipping the displacement and any initial holes in the view). For example, an offset of 2 for process 1 in Figure 2.10 is the position of the 8th etype in the file after the displacement.

A file handle is created by MPI_File_open and cleared by MPI_File_close. All operations on an open file reference the file through the file handle.

2.3.3 ROMIO architecture

ROMIO [TGL99b] is the most wide-spread implementation of MPI-IO standard and is developed at Argonne National Laboratory. ROMIO is part of several MPI distributions: MPICH [MPI95], OpenMPI [Edg04], MPI-HP [HP], MPI-NEC [NEC], MPI-SGI [SGI], etc. The architecture of ROMIO allows the virtualization of MPI-IO files on top of files of concrete file systems through ADIO. ADIO [TL96] is an abstract I/O interface, which can be easily specialized for specific file system implementations. The main goal of ADIO is to facilitate a high-performance implementation of new parallel I/O API for new file system. ADIO consists of a small set of basic functions for performing parallel I/O. Any parallel I/O API (including a file-system interface) can be implemented in a portable fashion on top of ADIO. In other words, ADIO separates the file system-dependent and file system-independent aspects involved in implementing an API. The file system-independent part can be implemented portably on top of ADIO. The file system-dependent part is ADIO itself, which must be implemented separately on each different system.

The ADIO interface contains typical functions for handling files: `open`, `close`, `fcntl`, `read`, `write`, etc. The read/write access operations may be individual or collective. For instance the two-phase collective I/O is implemented at this level. MPI-IO interface is mapped on ADIO functions.

The processing of MPI-IO operations can be controlled via the MPI API using file hints [CG99]. A file hint can affect how the MPI-IO library accesses the file. It can set buffer sizes, turn special optimizations on and off or provide specific parameters to each particular MPI-IO implementation.

2.4 Large-scale scientific data formats

Large-scale scientific data is often stored in scientific data formats such as netCDF and HDF. These storage formats are of particular interest to the scientific user community since they provide

multi-dimensional storage and retrieval. The most recent implementations of these data formats, such as HDF5 and pNetCDF (parallel netCDF), have been extended to support parallel data access, which is a key requirement for the data output for simulation codes on MPPs.

HDF Hierarchical Data Format (HDF) [HDF] is a scientific library, developed at NCSA, for storing, retrieving, analyzing, visualizing, and converting scientific data. The most popular versions of HDF are HDF4 and HDF5. Both versions store multidimensional arrays together with ancillary data in portable, self-describing file formats. HDF4 was designed with serial data access in mind, much like the current netCDF interface (described in next subsection). HDF5 is a major revision in which its API is completely redesigned and includes parallel I/O access. The support for parallel data access in HDF5 is built on top of MPI-IO, which ensures its portability.

The main projects that use the HDF library are the following. First, the OPeNDAP [CGS03] project is an open source project for implementing the DAP (data access protocol). The goal of OPeNDAP is to allow users to request data from remote sources, and to allow them to imagine the remote data to be stored in whatever format is most convenient for them. Second, HDF5 is a defacto standard in the scientific and engineering community including the NASA Earth Observing System project (HDF-EOS) of the NASA Earth Observatory [CDT06] . Finally, FLASH I/O is a smaller version of the FLASH code [FOR+00] that simply simulates FLASH's I/O patterns. The data domain is divided into blocks distributed across the processors. The benchmark uses the parallel HDF5 library for data I/O. FLASH I/O is one of the benchmark used in the evaluation of our architecture.

NetCDF The Network Common Data Form (netCDF) [RDED97] is a scientific library for atmospheric science applications. NetCDF intends to provide a common data access method to deal with a variety of data types that encompass single-point observations, time series, regularly spaced grids, and satellite or radar images.

The original design of the netCDF interface is proving inadequate for parallel applications because of its lack of a parallel access mechanism. In particular, there is no support for concurrently writing to a netCDF file. Therefore, parallel applications operating on netCDF files must serialize access.

To facilitate parallel I/O operations, in [LLC+03] the authors propose the design of a parallel API for concurrently accessing netCDF files. pnetCDF is implemented on top of MPI-IO, which is specified by the MPI-2 standard and is freely available on most platforms. The implementation underneath incorporates well-known parallel I/O techniques such as collective I/O to allow high-performance data access.

2.5 Parallel I/O optimizations

This section presents the most important contributions in parallel I/O optimizations.

2.5.1 Non-contiguous I/O

As discussed in the introductory section, many parallel scientific applications generate small granularity non-contiguous access patterns. This subsection reviews some non-contiguous I/O optimizations from the literature.

Data Sieving The I/O performance suffers considerably if applications access data by making many small I/O requests. In order to reduce the number of small requests for non-contiguous file regions, data sieving [TGL99a] reads the whole range between the minimum and maximum offsets of all regions and filters out the data of interest for read operations. For write operations, a read-modify-write process is needed. Data sieving pays off if the total size of the holes is sufficiently small, when compared with the useful data.

List I/O In order to enhance performance of non-contiguous I/O in PVFS, the authors proposed list I/O [CCL+02], a native version of non-contiguous I/O. The list I/O method is an optimization for high-performance file systems. In list I/O, the non-contiguous accesses are specified through a list of offsets of contiguous memory or file regions and a list of lengths of contiguous regions. List I/O reduces the number of I/O requests in a non-contiguous data access by describing multiple file regions in a single list I/O request.

Datatype I/O Datatype I/O method [CCL+03] is similar to list I/O with the difference that MPI data types are used instead of lists of offset-length tuples.

View I/O View I/O [IT03b] has detailed knowledge about parallel structure of a file and about the potential access pattern and exploits it in order to improve performance. The access overhead is reduced by using a strategy "declare once, use several times" and by file offset compaction.

2.5.2 Collective I/O

In MPI-IO data is moved between files and processes by issuing read and write calls. The data access routines may be individual or collective. Collective I/O techniques merge small individual requests from different compute nodes into larger global requests in order to optimize the network and disk performance. Depending on the place where the request merging occurs, one can identify two collective I/O methods. If the requests are merged at the I/O nodes the method is called *disk-directed I/O* [Kot94, SCJ+95]. If the merging occurs at intermediary nodes or at compute nodes the method is called *two-phase I/O* [dRBC93, Bor97]. Additionally, this section presents other collective I/O techniques, namely, partitioned collective I/O and split writing and hierarchical striping.

Disk-directed I/O* and *Server-directed I/O In disk-directed collective I/O technique [Kot94], I/O nodes gather I/O requests from all processes and then order block accesses in order to optimize disk seek time. Performance is likely to be much better than if I/O requests had been handled in the order of generation of the individual file accesses of different processes.

Panda[WSC+96] uses a server-directed I/O strategy in the implementation of the collective I/O operations. When compute nodes make collective I/O requests to Panda, a selected compute node (the master client) sends to a selected I/O node (the master server) a short high-level description of the in-memory and on-disk distributions for the arrays. The master server then provides all the other servers with the distribution information and each server independently plans how it will request or send its assigned disk chunks of the array data to or from the relevant clients.

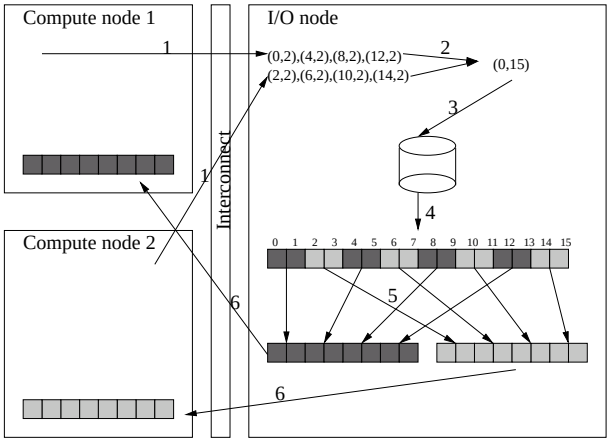

Figure 2.12: *The disk-directed collective.*

Two Phase I/O Two-phase I/O of ROMIO distribution performs the two steps illustrated in the Figure 2.13 for the collective write case: *shuffle phase* and *I/O access phase*. Four compute nodes shown in the upper part have previously declared views on the file depicted in the lower part of the figure. Compute node 0 "sees" only the dark gray bytes of the file, compute node 1 only the hashed, and so on. The access of each process is non-contiguous: for processor 0, the bytes 0, 1, 2, 3 of the view map on bytes 0, 4, 8, 12 of the file. In the shuffle phase, the data is gathered in contiguous chunks at a subset of the compute nodes acting as *aggregators*. In this example we have used two aggregators (compute nodes 1 and 2). In general, the number of aggregators can be decided by the user with a hint (by default all compute nodes are aggregators). In the first part of shuffle phase, the interval to be accessed (16 bytes between offsets 0 and 15) in split among the two aggregators: (0,7) and (8,15). Then the (offset, length) lists, which correspond to the mapping between each view and the file, are sent from all compute nodes to aggregators. For instance, compute node 0 sends (0,1), (4,1) to aggregator 0 and (8,1), (12,1) to aggregator 1. Finally, the view data is transferred to the aggregators and scattered into contiguous chunks by using the offset length lists. In the access phase, the contiguous chunks are transferred to the file system. The access phase is fast, because only contiguous transfers are requested to the file systems. The I/O access phase is implemented through an ADIO function call, which in turn calls file system access functions.

We note that, at each access, the compute nodes must agree on the interval splitting and must build and send the (offset,length) lists to the aggregators. These operations involve collective communication and synchronization. Additionally, for small granularities, the offset-length lists may become substantially large.

Data shipping Data shipping [PTH+01, PTH+00] is a collective optimization implemented in the GPFS library. Data shipping disables local caching of a file and uniquely binds each file block in a round-robin manner to one I/O agent. All subsequent read and write operation on the file go through the I/O agents, which ship the requested data between the file system and the appropriate processes. I/O agents play the same roles as the aggregators in the ROMIO's two-phase collective I/O.

For write operations, each task distributes the data to be written to the I/O agents according

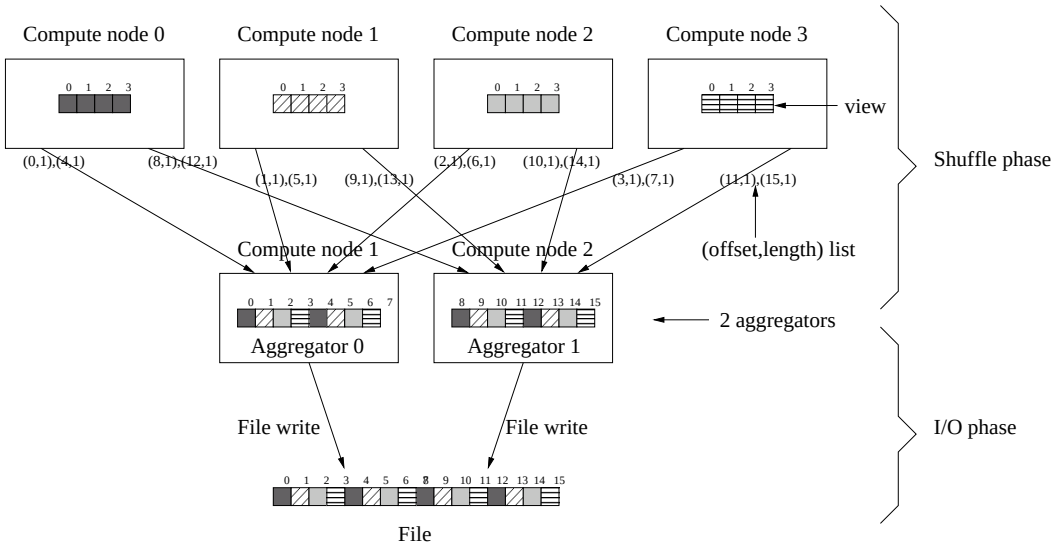

Figure 2.13: *Two phase I/O write example. In this case, the first phase is communication and the second phase is I/O.*

to the binding scheme of GPFS blocks to I/O agents. I/O agents in turn issue the write calls to GPFS. For reads, the I/O agents read the file first, and ship the data read to the appropriate tasks.

Partitioned collective I/O In [YV08], the authors introduced a novel technique called partitioned collective I/O (ParColl). ParColl augments the original two-phase collective I/O (explained previously) with new mechanisms for file partitioning, I/O aggregator distribution and intermediate file views. Through these mechanisms, a group of processes and their targeted file are consistently divided into a collection of small subgroups, each performing I/O aggregation in a disjoint manner. File consistency is maintained through intermediate file views when necessary. Together, these mechanisms greatly reduce the cost of global synchronization.

Split writing and hierarchical striping In order to mitigate striping overhead and benefit from the collective I/O accesses on Lustre [Wei07] (previously described), the authors proposed two techniques: split writing and hierarchical striping. In split writing, a file is created as separate sub-files, each of which is striped to only a few storage devices. They are joined as a single file at the file close time. In the second technique, hierarchical striping builds on top of split writing and orchestrates the span of sub-files in a hierarchical manner, in order to avoid overlapping and achieve the appropriate coverage of storage devices. Together, these techniques can avoid the overhead associated with large stripe width, while still being able to combine bandwidth available from many storage devices.

2.6 File access latency hiding optimizations

File latency hiding is an optimization technique to eliminate time to wait for file accesess, where the basic idea is to overlap computation, communication, and I/O.

2.6.1 Write-back

A significant amount of prior research has focused on hiding the I/O latency of parallel applications using write-back methods.

MTIO. More et al. implemented a multi-threaded MPI-based I/O library called MTIO. MTIO [MCFX97] uses a single I/O thread to perform collective I/O in the background. The authors reported an overlap of up to 80% between computation and I/O for the benchmark program on the IBM SP2. Dickens et al. also studied the use of threads to improve the collective I/O performance of applications [DT99]. However, instead of performing the entire collective I/O in the background, the authors suggest overlapping only the actual write phase with the foreground computation and communication.

Active Buffering. Active buffering is an optimization for MPI collective write operations [MWLY03]. It buffers output data locally and uses an I/O thread to perform write requests in background. Using I/O threads allows to dynamically adjusting the size of local buffer based on available memory space. Active buffering creates only one thread for the entire run of a program, which alleviates the overhead of spawning a new thread every time a collective I/O call is made. For each write request, the main thread allocates a buffer, copies the data over, and inserts this buffer into a queue. The I/O thread, running in background, later retrieves the buffers from the head of the queue, issues write calls to the file system, and releases the buffer. Although write-behind enhances parallel write performance, active buffering is applicable if the I/O patterns only consist of write operations. Lacking of consistency control, active buffering could not handle the operations mixed with reads and writes as well as independent and collective calls.

Collective caching. In [LCC+05], the authors present a user-space implementation that uses an I/O thread in each client process to handle the local and remote access to the cache data. The thread catches read()/write() system calls from applications and determines whether the I/O request should go to the file servers or the caching sub-system.

2.6.2 Prefetching

I/O prefetching is a technique for improving file access performance by issuing file data requests in advance [CKV93, KTP+96, KE93]. Informed prefetching and caching [PGG+95] leverages application disclosed access pattern in order to decide cost-efficient trade-offs between prefetching and caching policies. Chang and Gibson [CG99] propose an automatic speculative prefetching technique. When incurring a cache read miss, the application uses the CPU spare time in order to speculatively continue the execution and generate prefetch requests for potential future read accesses. PC-OPT [KV02] is an off-line prefetching and caching algorithm for parallel I/O systems. When PC-OPT has a priori knowledge of the entire reference sequence, it generates a schedule of minimal length. In [BCS+08] the authors proposed an I/O prefetching method based on I/O pattern signatures. Signatures are used by the application to guide prefetching. A prefetching thread reads I/O signatures of an application and adjusts them by observing the I/O pattern at runtime. [CBS+08b] propose a pre-execution prefetching approach to improve the I/O access performance of parallel applications. Prefetching methods presented are based on pre-execution knowledge.

2.7 Characterization of I/O accesses in HPC environments

The characterization of data-intensive HPC workloads has been the topic of many and on going studies [SR97, SR98, SAS08, BCS+08].

Applications exhibit repetitive behaviour when a loop or a function with loops issues I/O requests [BCS+08]. More et al. classifies I/O access patterns either with repetitive behaviour or without (i.e., pattern occurs only once). When I/O access patterns are repetitive, caching and prefetching can effectively mask their access latency [PSK08]. Additionally, the sequence of file locations accessed has been discovered to be contiguous, non-contiguous, or a combination of both. Non-contiguous accesses refer to gaps (or strides between successive file offsets) in accessing a file. These gaps can be of fixed size or variable size. Variable size gaps can follow a pattern of two or more dimensions (2d or kd). Some I/O accesses have no regular pattern, where the strides are random.

In the previous mentioned studies, the researchers investigated several I/O intensive parallel scientific applications, such as MADBench2 [BOSS07], S3D [J. 09], and FLASH3 code [htt09a, FKL+08], to mention a few.

MADBench2 is a benchmark derived from a cosmology application that analyzes Cosmic Microwave Background data sets. MADBench2 spends approximately 81% of its total run time within MPI-IO read or write calls. Its most unusual characteristic is that it spends a significant portion of its time overwriting data. The out-of-core algorithm for MADBench2 resulted in each process alternating between read and write several times within the same file (7 per process). MADBench2 offers little opportunities for MPI-IO optimization or tuning, since all data is accessed contiguously by using named datatypes and independent access.

The S3D is a parallel turbulent combustion application using a direct numerical simulation solver developed at Sandia National Laboratories. A checkpoint is performed at regular intervals, and its data consists primarily of the solved variables in 8-byte three-dimensional arrays, corresponding to the values at the three-dimensional Cartesian mesh points. S3D uses the pnetCDF collective interface, but through the use of MPI-IO hints we were able to evaluate both collective and independent I/O. Application performs one million 8-byte writes using independent I/O mode. On Blue Gene, this workload is a recipe for disaster due to I/O forwarding latency and the lack of write caching at compute nodes [BOSS07].

The FLASH code is a modular adaptive mesh code used for simulating compressible reactive flows in astrophysical environments, primarily focused on the deflagration and detonation of type Ia supernovae. While various physics applications may be run with the FLASH code, the I/O pattern for these applications is largely the same and consists of writing grid variables for checkpoint/restarting and smaller single precision plotfiles for visualization and analysis.

The above mentioned data-intensive parallel scientific applications show characteristics common to other previously studied workloads [NKP+96, SR97].

Individual compute nodes often access files non-contiguously. Non-contiguous accesses cause costly network transfers, and, therefore, have a negative influence on performance. However, with different physical partitioning approaches, the contiguous access of each compute node may translate into contiguous disk access.

Compute nodes frequently access a file by using interleaved patterns. However, the global access pattern, composed from the individual accesses of compute nodes, is contiguous. If they occur approximatively at the same time (i.e. temporal locality), the global disk access pattern is

sequential, and therefore optimal. Otherwise, unnecessary seek times may drastically affect the application performance. This problem is addressed by different collective I/O operations designs and implementations [dRBC93, Kot94].

The investigated applications generate a high number of small I/O requests. To a large extent this was the result of the non-contiguous and interleaved accesses, as described earlier. Additionally, parallel scientific applications use nested strided access pattern. This access type occurs with multidimensional arrays that are partitioned across compute nodes.

2.8 Summary

This chapter presented a complete vision of the problematic of the data-intensive I/O environments, included the following points: most recent I/O architectures used on massively computing, the more extended parallel file systems, and different I/O optimizations applied on those.

The traditional solution based on the direct connection of storage systems have evolved toward solutions based on the use of a network of high speed interconnects linking a set of storage systems with computers, which access the data stored there. Connecting the system directly to a storage network storage system allows to add more storage in a scalable manner.

The tendency to virtualize storage (providing a logical view to users free of physical solution actually used) facilitates the administration, offering a greater flexibility.

Chapter 3

Generic design of the parallel I/O architecture

Many scientific applications use parallel I/O to meet the low latency and high bandwidth I/O requirement. Among many available parallel I/O operations, collective I/O is one of the most popular methods when the storage layouts and access patterns of data do not match. To achieve better I/O performance, client-side caching is often considered as a scaling technique [CGL97, LCC+05]. It places a replica of repeated access data in the memory of the requesting processors such that successive I/O requests to the same data can be carried out locally without going to the file servers.

There has been past research on automating the overlapping of I/O operation with computation and communication in the foreground, also known as split-collective I/O. However, this approach does not overlap all the tier of an application simultaneously. Instead, it first sequentializes two phases of the application and then overlaps it with the third phase of the program. Dickens et al. also studied the use of threads to improve the collective I/O performance of applications [DT99]. However, instead of performing the entire collective I/O in the background, the authors suggest overlapping only the actual write phase with the foreground computation and communication.

The motivation of this work is to enhance the utilization of the storage and network resources in a generic architecture. We mainly target the optimization of both contiguous and non-contiguous file accesses with small and medium granularities, such as the one exhibited by a significant class of scientific applications [NKP+96, SR97].

This thesis addresses the problem of finding a general architecture for I/O operations in large parallel computing environments such as clusters or supercomputers. The variety of solutions and architectures of the field of study revolves around models and abstractions that share not only goals, but also certain features. One of the aims of this thesis is to provide a generic parallel I/O architecture that can be thought of as a common denominator for the shared features of state-of-the-art parallel I/O systems, while preserving portability and application transparency.

The rest of this chapter is organized as follows. We start by describing the logical and physical components of the architecture and discuss how they map on real architecture of clusters and supercomputers. The details of each design are provided in Chapters 4 and 5 respectively. The remainder of this chapter is dedicated to the common denominator of both cluster and supercomputer architectures, namely *generic client-side file cache management*.

3.1 Generic parallel I/O architecture

Our proposed solution is based on a multi-tier hierarchy as depicted in Figure 3.1. The generic parallel I/O architecture is organized on six tiers: application, application I/O forwarding, client-side cache, aggregator I/O forwarding, I/O node-side cache and storage. In the figure we show the components of our generic parallel I/O architecture and on the margins, we depict how the logical components map on physical nodes.

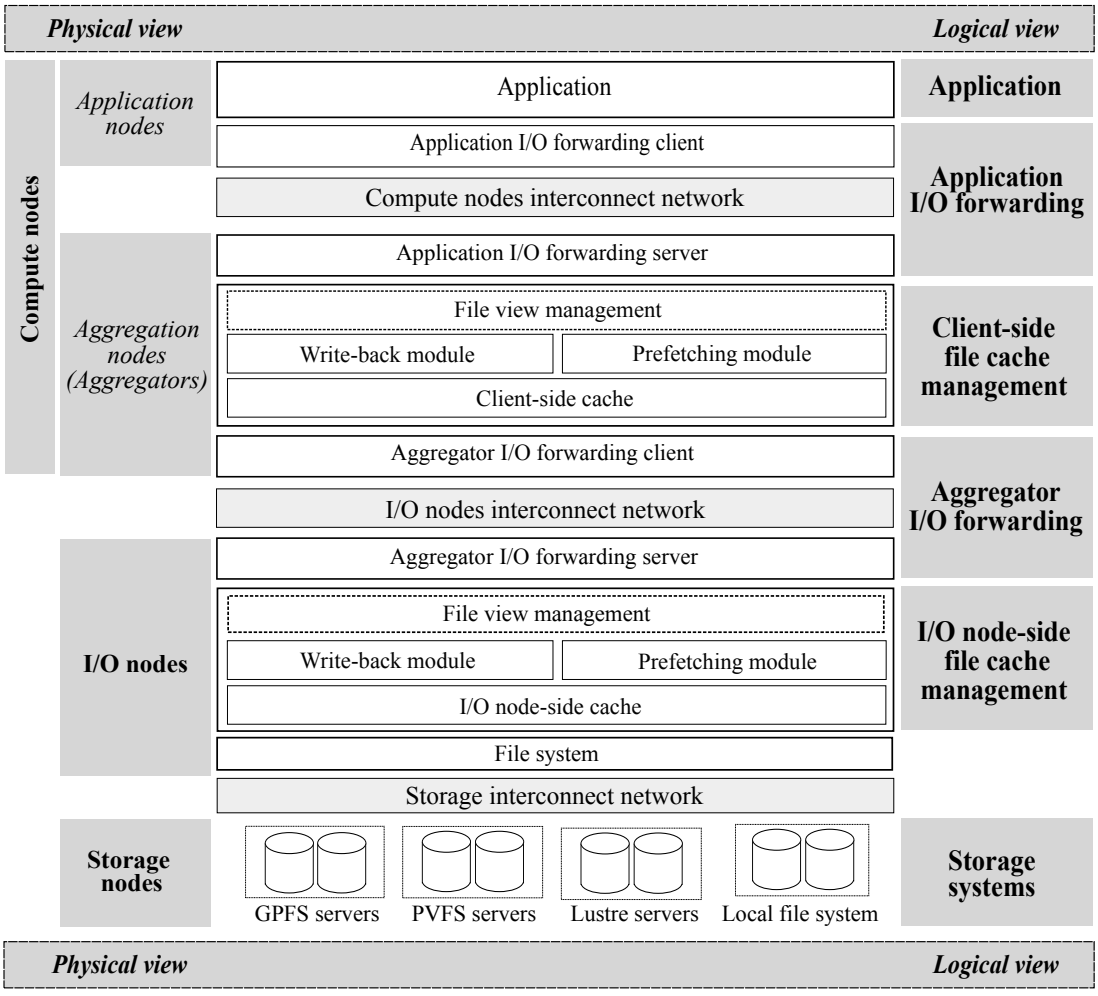

Figure 3.1: *Generic parallel I/O architecture organized on six tiers: application, application I/O forwarding, client-side cache, aggregator I/O forwarding, I/O node-side cache and storage. Dotted boxes represent optional modules, which are client and I/O node-side file view management modules.*

Application tier. MPI applications run on compute nodes and access the generic parallel I/O architecture through file interfaces such as MPI-IO or POSIX. In order to use our architecture, the application does not need to be modified.

Application I/O forwarding tier. Application I/O forwarding tier resides on all application nodes and has the goal of forwarding the file accesses to the next tier through the compute nodes interconnect network. The forwarding is done on-demand, when the application issues the file access.

Client-side file cache management tier. The client-side file cache module resides on aggregegator nodes, namely aggregators. Aggregators are compute nodes, which are in charge of merging small file access requests and performing the file access on behalf of several processes of the application. Aggregators manage in cooperation the client-side distributed cache. Unlike in ROMIO, aggregators participate not only in collective I/O operations but also in independent I/O. A configurable number of aggregators have an associated I/O node, which performs I/O operations on behalf of the compute nodes.

The application accesses the client-side cache through the application I/O forwarding tier. The objective of client-side file cache module is to provide the management of a file cache close to the applications and to offer efficient transfer methods between the applications and I/O subsystem. A distributed file file cache is stored on the local memory of compute nodes and is managed in cooperation by them. The main tasks of the module are buffer management and asynchronous data staging, including write-back and prefetching modules. This module is network- and file system-independent and is common for both cluster and supercomputer architectures. The client-side cache management tier absorbs the writes of the applications and hides the latency of accessing the I/O nodes over interconnect networks.

Aggregator I/O forwarding tier. Aggregator I/O forwarding tier resides on all aggregation nodes and has the goal of forwarding the file accesses to the *I/O node-side file cache* through the I/O nodes interconnection network. The forwarding is done on-demand, when the aggregator issues the file access. The aggregator I/O forwarding server running on each I/O node manages the I/O node-side file cache management layer.

The aggregator I/O forwarding server receives aggregator requests from the aggregators and serves them from the cache. The communication with the compute nodes may be decoupled from the file system access, allowing for a full overlap of the two operations.

I/O node-side file cache management tier. The I/O node-side file cache management tier resides on all I/O nodes. The objective of I/O node-side file cache module is to provide the management of a file cache close to the storage system and offer efficient transfer methods between compute nodes and the storage system. The I/O node-side file cache management tier absorbs the blocks transferred asynchronously from the client-side cache through the aggregator I/O forwarding client, and hides the transfers between the I/O nodes and file systems. An I/O thread is responsible for asynchronously accessing the file systems by enforcing the generic write-back and prefetching policies (described in Section 3.3.3).

This module is specific for each physical I/O architecture, namely for clusters and supercomputers. However, in both cases this module virtualizes the file system access through specific mechanisms. Depending on the physical I/O architecture, this module may reside on the main memory of either compute nodes responsible of issuing file system requests (either to a local or to a shared file system) or on dedicated I/O nodes.

Storage system tier. The file system components run on dedicated file servers connected

to storage nodes through the storage interconnection network. The storage system provides the persistent storage service and can be any file system providing a VFS (virtual file system) interface. In this thesis we have employed both local file systems virtualized through our architecture, and parallel file systems installed in clusters and supercomputers such as GPFS, PVFS, and Lustre.

3.2 Generic data staging

Our generic data staging solution is based on the multi-level file cache hierarchy from Figure 3.1. The first level of distributed file caching, similar to [LCC+05], is deployed on compute nodes to optimize the I/O requests for parallel file systems. The distributed cache system completes the requested I/O operation by caching data or calling the corresponding system calls. The result of this I/O request is returned to the application node and then computation at the application node is resumed. The I/O servers manage the second level of distributed caching at I/O nodes. File blocks are mapped to I/O servers in a round-robin fashion and each server is responsible for transferring its blocks to and from the persistent storage.

In order to hide the latency of file accesses, we propose a data staging method on both client- and I/O node-sides. We describe the two-level hierarchy consisting of client-side and I/O node-side caching. The client-side file cache management module coordinates the data staging between application and I/O nodes through the I/O nodes interconnect network, while the I/O node-side file cache management module coordinates the data staging between I/O nodes and storage through the storage interconnect network.

Multi-level write-back After a compute node issues a file write, data are pipelined from compute nodes through the I/O nodes to the file systems. An application write request is transferred by application I/O forwarding tier to the client-side cache management tier. The cached file blocks are marked dirty, the application is acknowledged the successful transfer and is allowed to continue. The responsibility of flushing the data from client-side cache to I/O node over the I/O network belongs to a write-back module. On the I/O node a write-back module is in charge of caching the file blocks received from the compute nodes and flushing them to the file system over the storage network (e.g. Myrinet).

The write-back policies are based on a high/low water mark for dirty blocks. The high and low water marks are percentages of dirty blocks. The flushing of dirty blocks is activated when a high water mark of blocks is reached. Once activated the flushing continues until the number of dirty blocks falls below a low water mark. Blocks are chosen to be flushed in the Least Recently Modified (LRM) order.

In order to efficiently hide the latency, coordination along the pipeline is necessary. We highlight two important aspects. First, the coordination has to take into account the application requirements. For instance, in parallel applications processes frequently write shared files in non-overlapping manner, showing a good inter-process spatial locality. For these applications the high and low water marks should be sized in such a manner that makes improbable the transfer of incompletely written blocks. On the other hand, blocks of files written by sequential applications may be immediately flushed. Second, the coordination must take into consideration the network characteristics and loads.

Multi-level prefetching The file read pipeline involves transfers from storage through storage nodes and I/O nodes toward the compute nodes. Our prefetching solution proposes two prefetching modules on the compute node and I/O node, each of which enforcing its own prefetching policy. The prefetching mechanism is leveraged in both cases by an I/O thread.

The client-side prefetching policy is based on two main parameters: mapping of files to aggregators and views. If no view is declared, the view is by default the whole file. The application process sends the view to all file aggregators after declaration, as described in Section 3.3.1. Any time an on-demand read request misses the client-side cache, it is issued and, while served, the subsequent file view offsets are mapped on the file and a prefetching request from the I/O node is issued. A configurable number of prefetching requests can be generated. The views bring the advantage of generating any type of prefetching pattern, including the common sequential, simple and multiple-strided.

The I/O node prefetching policy is based on the mapping of aggregation nodes on the I/O nodes. Each I/O node prefetching module is aware of the aggregators and on the mapping of files on aggregators. Whenever an aggregator makes an on demand request, the next file blocks mapped on the aggregators are computed and prefetch requests are issued. Note that the prefetching requests of the client-side module are seen on the I/O node as on-demand file requests. Therefore, the I/O node-side prefetching module plays in this case the role of further activating the next levels of prefetching pipeline.

In Section 3.3.3 we present the generic write-back and prefetching operations. In subsequent chapters we will describe how these generic operations map on concrete architectures.

3.3 Generic client-side file cache management

In this section we describe in details the common components of our generic architecture for both clusters and supercomputers, namely generic client-side file cache management. In the following section we depict the file view management, data staging, and file cache modules.

3.3.1 File cache module

In order to hide the latency of file accesses, we have proposed a distributed file cache. The distributed file cache resides in the file cache module. Our distributed file cache mechanism consists in putting in common a fraction of the local memory as cache buffers. The cache is managed through the cooperation of all compute nodes (using only a subset of processes is also possible). Distributed file cache stores in memory collective buffers. Collective buffers are used by compute nodes for reordering and gathering I/O requests in order to improve the I/O performance. In order to avoid cache coherency problems, we propose a single-copy distributed cache. At any time, there is unique copy of the data in the distributed cache as described in Section 3.3.4.

For write, when an aggregator receives a message from a compute node, it checks if it has already cached its corresponding collective buffer. If so, the data is scattered by the view data type to the collective buffer. Otherwise, the buffer is first read from the file system (for an existing file) and then the scatter operation is performed.

For read, the collective buffer is read from the file system at the first request of a compute node. Subsequent accesses find the data in the cache. It can be noticed that this approach targets

the spatial locality characteristic of the parallel scientific applications [NKP+96, SR98].

There are three main differences between our proposed solution for distributed file caching and collective buffers of two-phase I/O. First, if the aggregators have the data cached, only one network transfer is necessary, while in two-phase I/O at least two transfers are performed (one for shuffle and one for file I/O). Second, in two-phase I/O, the collective buffers are not reused across different collective operations. Consequently, workloads that show temporal locality cannot take advantage of the data already cached. In contrast, in our solution, the collective buffers are stored in the local cache of the aggregators and can be reused across different collective I/O operations. Third, the data staging strategies allow overlapping the computation and I/O, by gradually transferring the data from the collective buffer cache to the I/O nodes.

In order to access the file system, aggregators employ the aggregator I/O forwarding module. The amount of bytes write/read operations depends on collective buffer striping factor (how the file intervals are statically mapped onto aggregators). Different striping factors determine performance according to the file system accessed.

3.3.2 File view management module

We propose a generic access method for the file view management, namely view-based I/O. View-based I/O is a file system independent I/O optimization based on file views. View-based I/O leverages the MPI-IO file view mechanism, for transferring view description information at aggregators at view declaration. In this way, view-based I/O avoids the necessity of transferring large lists of offset-length pairs at file access time as the present solution of two-phase I/O. Additionally, this approach reduces the cost of scatter/gather operations at application compute nodes. View-based I/O reads can take advantage of collective buffers managed by aggregators, which, unlike in two-phase I/O are cached across collective operations. A collective read following a previous read or write, may find the data already at the aggregator.

In the rest of this section, we present view-based I/O and describe some design details such as file views and data access operations.

Overview

As two-phase I/O, view-based I/O is a file-system independent I/O optimization. However, there are several differences between the two methods. We will discuss these differences after presenting a short overview of view-based I/O.

View-based I/O consists of the following steps. Initially, all compute nodes send the view MPI data type to all aggregators, where it is kept for the future accesses. The MPI data type is nothing else than the compact representation of all the classes of file accesses that can be performed for its associated view. Therefore, all the offset/length lists can be generated from this data type and must not be sent for each access as in two-phase I/O. More details about the view management design of view-based I/O are given in Section 3.3.2.

View-based I/O access phase is designed starting from the assumption that the file is mapped following round-robin pattern on the aggregators, with a stripe that can be set by means of an MPI hint and, whose default value is the same as the collective buffer size for two-phase I/O. This avoids the file interval splitting step from two-phase I/O.

At access time, a compute node sends to the aggregator only the view data plus the borders of

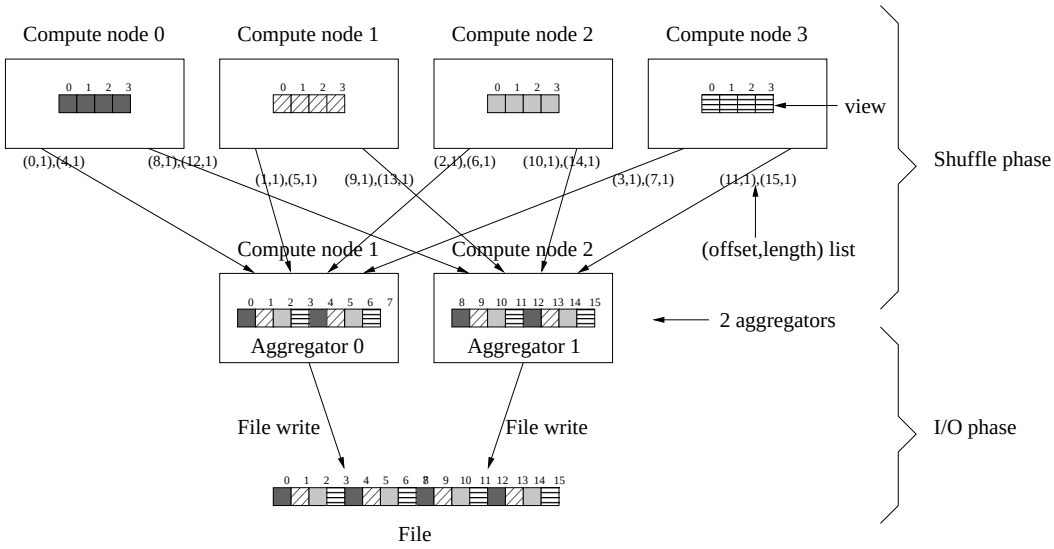

Figure 3.2: *Mapping view and lists of offset/lenght of two-phase I/O.*

the view interval corresponding to the aggregator (as in Figure 3.3a). For instance, in the example from Figure 3.2, the file is assigned statically 2 aggregators by using a stripe of 8. Therefore, the compute node 0 sends to the aggregator 0 (compute node 1) the start and end offsets of the view, (0,1) and the corresponding data and to the aggregator 1 (2,3) and the corresponding data. Each aggregator, uses the previously stored data type to scatter the data into the collective buffer.

Each aggregator manages a pool of collective buffers as shown in Figure 3.3b. This pool of collective buffers corresponds with the distributed file cache. The size of this pool (the number of available collective buffers in the distributed file cache) may be controlled by a user hint. Whenever all the collective buffers are used and a new one is requested, a LRU replacement policy is used. In our design, a file is logically divided into equally sized blocks that are the minimal caching unit. As we show in Figure 3.3c, the collective buffers are flushed when the file is closed or when the write-back policy is activated (more details are given in Section 3.3.3).

In summary, the main differences between two-phase I/O and view-based I/O are the following:

- At view declaration, view-based I/O sends the view data type to aggregators, while two-phase I/O stores it locally at the application nodes.
- View-based I/O assigns statically the file domain to aggregators, while two-phase I/O is dynamic.
- At access time, two-phase I/O sends the offset-lists to the aggregators, while view I/O transfers only the view access interval boundaries.
- The collective buffers of view-based I/O are cached across collective operations into the distributed file cache. A collective read following a write, may find the data already at the aggregator.
- The distributed file cache of view-based I/O is managed by a dedicated data staging thread, which asynchronously stages the data between the client-side cache and I/O node-side cache

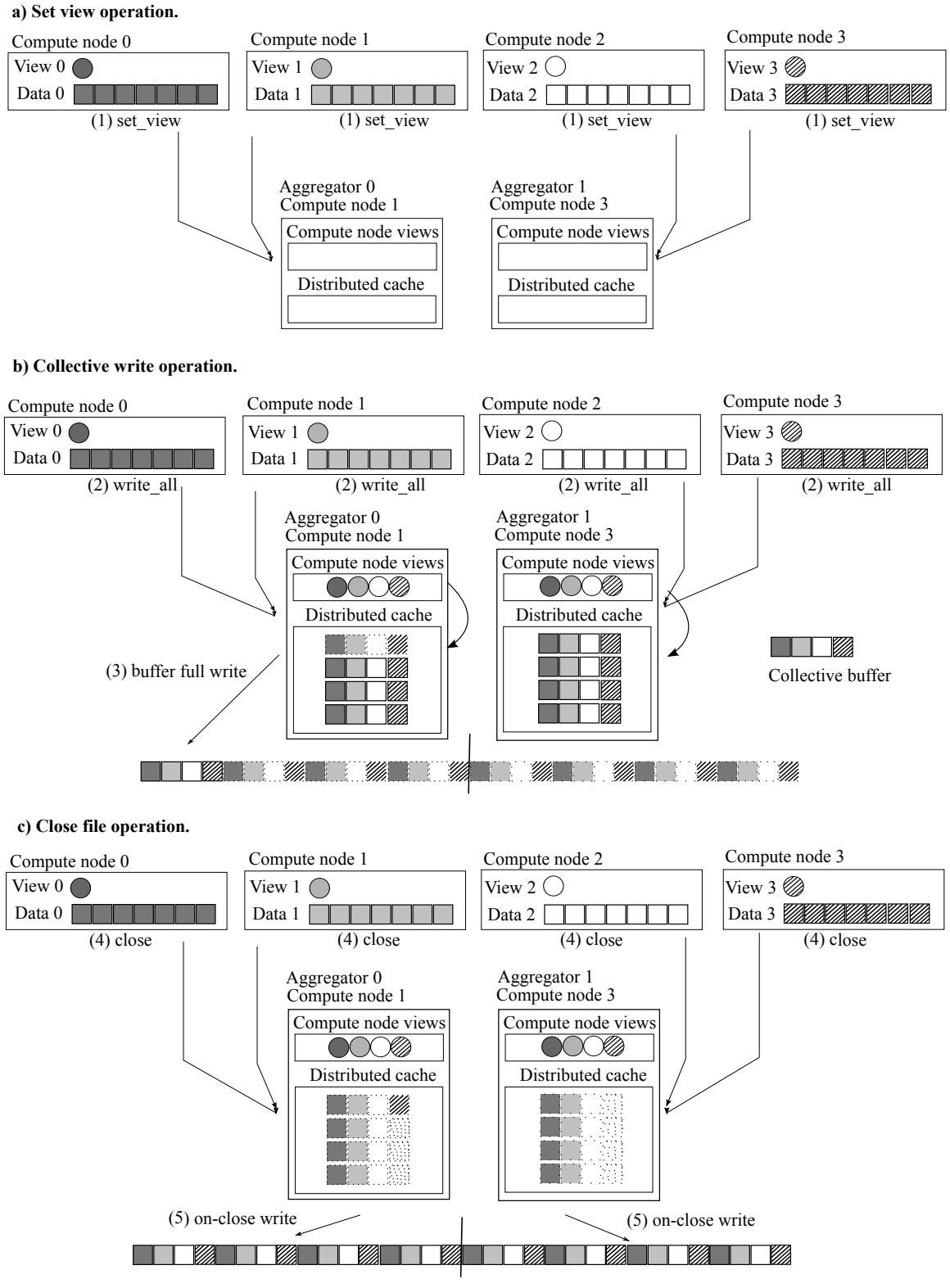

Figure 3.3: *View-based I/O data access operations steps: a) Compute nodes send to the aggregators the view, b) each aggregator manages a pool of collective buffers, c) the collective buffers are flushed when the file is closed or when the write-back policy is activated.*

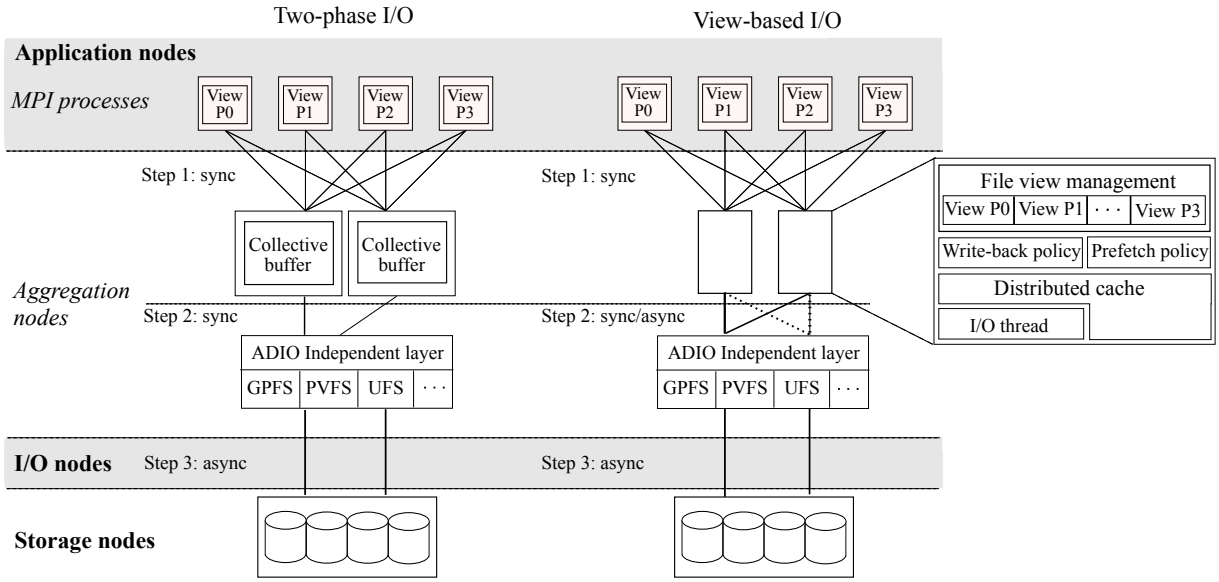

Figure 3.4: *Comparison of two-phase I/O and view-based I/O.*

or file system cache, through the aggregator I/O forwarding layer. The write requests are blocking only when the buffer cache is full and dirty. The file data is fully committed to the next cache level, when the file is closed.

- The design includes a prefetching module. Views compactly represent access patterns and that they may be used as hint of future access.

Design and implementation

View-based I/O resides in the ADIO layer and is conformed to all portability criteria imposed by the MPI standard, making it file-system independent and hence available on all platforms. The proposed design needs to modify open, close, write, read and set view ADIO primitives.

The existing MPI-IO optimizations are not affected by the modifications in ROMIO. A hint allows the user to select methods implemented either in the original MPI distributions or view-based I/O. The following subsections describe the building blocks of our approach: views, data access operations, collective buffers, and the data staging methods proposed.

Views We propose a novel view mechanism alternative to the one from ROMIO. Unlike in two-phase I/O, upon view declaration, each compute node decodes the view data type. This is achieved by a recursive top-down traversal of the data type tree, which reconstructs the steps employed by the original application for data type creation. This data type structure is packed and transferred to all aggregators, which store data belonging to the file. Upon reception, the aggregators reconstruct the original application-view data types and store them for subsequent use.

Data access operations For both read and write operations, each compute node manages several task queues, one for each aggregator. For write operations the following steps take place.

1.- Each node maps the view interval onto aggregators. In the example from Figure 3.2, compute node 0 maps the interval (0,1) onto aggregator 0 and the interval (2,3) on aggregator 1. For each of these intervals a communication task is created and placed in the queue of its corresponding aggregator.

2.- View-based I/O uses a network buffer for performing the communication tasks. If data from several communication tasks fit in same network buffer, they are coalesced and sent through only one MPI call. The order in which the compute nodes send data to the aggregators is guided through a parallel I/O scheduling policy [JSWB97]. It tries to increase parallelism (by trying to enforce that each compute node sends data to a different aggregator). The aggregators receive the data from the compute nodes and scatter them into the collective buffers.

3.- Finally, when replacing a collective buffer from the collective buffer pool, or when the file is closed, the collective buffers are written to the file system.

For read operations, the following steps are performed.

1.- As for write, the same mapping and task creation operations are performed.

2.- The read requests are sent to the aggregators. In general case each aggregator receives read requests from several compute nodes. The aggregators gather the data from the collective buffers and send them to the corresponding compute node. As for write, the order in which the aggregators deliver the data to the compute nodes is decided through a parallel I/O scheduling policy.

3.- Finally, upon reception, the compute nodes finish the tasks from the queues by transferring the data into user buffers.

3.3.3 Data staging modules

In the following subsections we present the client-side write-back and prefetching policies, which are common for both cluster and supercomputer architectures. Specific purpose I/O node-side data staging policies for cluster and supercomputer architectures are presented in chapters 4 and 5, respectively.

The Table 3.1 contains a description of the variables and routines employed in the subsequent description of the data staging policies.

Table 3.1: *Description of variables and functions used by algorithms.*

Variable	**Description**
b	block indentifier
left_view_offset and *right_view_offset*	left and right view offsets
view_table	table of views (MPI data type)
CollBufferPool	collective buffer pool
PrefetchBufferPool	prefetch buffer pool
MAPPED_FILE_BLOCK_SET	set of blocks on which a view maps
PREFETCH_BLOCK_SCHEDULE	set of blocks scheduled for prefetching
Function	**Description**
full(BufferPool)	returns true if the buffer pool is full
insert(b,BufferPool)	inserts block *b* into the buffer pool *BufferPool*
remove(b,BufferPool)	removes block *b* from buffer pool *BufferPool*
replacementBlock(CollBufferPool)	chooses a block for replacement based on LRU policy
chooseFlushBlock(CollBufferPool)	chooses a block for flushing based on LRM policy
copyBlock(b,BufferPool1, BufferPool2)	moves block *b* from *BufferPool1* to *BufferPool2*
map(view_table[mpi_rank],left_view_offset,right_view_offset)	maps the view access interval onto file blocks
dirty(b)	returns true if the block is dirty
write(b)	writes the content of block *b* to the file system
read(b)	reads the content of file block *b* from the file system
scatter(netbuf, b,left_view_offset,right_view_offset,view)	scatters data from the network buffer into the collective block b
gather(netbuf, b,left_view_offset,right_view_offset,view)	gathers data from the collective block into the network buffer
predict_access(view_table, MAPPED_FILE_BLOCK_SET)	predicting future block accesses
gpfs_prefetch_finished(b)	returns if prefetching has finished for block *b*
activate_gpfs_prefetching(SCHEDULE, PrefetchBufferPool)	releases the prefetched blocks and schedules the next blocks

Client-side write-back

File write-back operation consists of the following steps. At view declaration, the compute node view (defined as MPI data types) is sent by all compute nodes to all aggregators. The aggregators store views in the *file view management layer* in a vector structure indexed by the MPI rank. At access time, a compute node sends to all aggregators only the view data plus the boundaries of the file access interval of the view: left view offset (*left_view_offset*) and right view offset(*right_view_offset*). The distributed file cache is represented by the *CollBufferPool* variable.

Subsequently, each aggregator runs Algorithm 1 for each compute node request received in the network buffer (line 1). First, the set of blocks to be written (*MAPPED_FILE_BLOCK_SET*) is calculated by mapping the view access interval onto file blocks through the previously stored view (line 2). For each block *b* from *MAPPED_FILE_BLOCK_SET* (line 3), if the block is not in the collective buffer pool (line 4) and if the buffer pool is full (line 5), a collective buffer *repl_b* is chosen for replacement by a LRU policy (line 6), flushed to the file system (line 8) if dirty, and removed from the pool (line 10). Finally, *b* is inserted into the collective buffer pool (line 12). At this point, data are scattered from the network buffer into *b* (line 14) and *b* is marked as dirty (line 15).

Algorithm 1 Write-back: Aggregator-side

```
1:  recv(netbuf, mpi_rank)
2:  MAPPED_FILE_BLOCK_SET ← map(view_table[mpi_rank], left_view_offset,
    right_view_offset)
3:  for all b in MAPPED_FILE_BLOCK_SET do
4:    if b not in CollBufferPool then
5:      if full(CollBufferPool) then
6:        repl_b ← replacementBlock(CollBufferPool)
7:        if dirty(repl_b) then
8:          write(repl_b)
9:        end if
10:       remove(repl_b, CollBufferPool)
11:     end if
12:     insert(b, CollBufferPool)
13:   end if
14:   scatter(netbuf,b, left_view_offset, right_view_offset, view)
15:   set_dirty(b)
16: end for
```

The write-back policy is enforced in an I/O thread shown as Algorithm 2 at the aggregators. The I/O thread is activated when a high water mark of dirty blocks is reached. While the number of dirty blocks is higher than a low water mark, the thread flushes buffers to the next cache level by a Last Recently Modified (LRM) policy. Finally, the I/O thread goes to sleep until the high water mark is reached again.

This write-back policy brings at least two main advantages. First, the last block inserted to the cache will be selected last for flushing. This allows taking advantage of the inter-process spatial locality characteristic of the parallel scientific applications. Second, it allows overlapping

of computation and I/O, by gradually transferring data from the collective buffer cache to I/O nodes. This approach distributes the cost of file accesses over the computation phase, addressing one more characteristic of scientific applications, namely the alternation of computation and I/O phases.

Algorithm 2 Write-back: I/O thread-side

```
1: while no_of_dirty_buffers < high_water_mark do
2:     sleep
3: end while
4: while no_of_dirty_buffers > low_water_mark do
5:     b ← chooseFlushBlock(CollBufferPool)
6:     write(b)
7: end while
```

Client-side prefetching

The prefetching policy is enforced at the aggregators and consists of: deciding which blocks to prefetch (prediction), monitoring the block prefetching, moving the prefetched blocks from *prefetch buffer pool* to the *collective buffer pool*. Prefetching is based on MPI views, which reflect potential future access patterns. As described in the previous section the views are stored at aggregators at view declaration. Aggretators manage a prefetch buffer pool of a few buffers with the size of a collective buffer.

At access time, a compute node sends to all aggregators only the extremities of the file access interval of the view: left view offset (*left_view_offset*) and right view offset (*right_view_offset*).

Subsequently, each aggregator runs Algorithm 3 for each compute node request. First, the set of blocks to be read from *MAPPED_FILE_BLOCK_SET* is calculated by mapping the view access interval onto file blocks through the previously stored view (line 1). For each block *b* from *MAPPED_FILE_BLOCK_SET* (line 2), if the block is not in the collective buffer pool (line 3), it is inserted (line 11), after replacing a block if the collective buffer pool was full (lines 4-10). If *b* has been already read into prefetching buffer pool (line 12), i.e. has been scheduled for prefetch previously, the future access is removed from prefetching buffer (line 13). If *b* has not been scheduled for prefetching, it is read directly from the I/O node-side module (line 15). At this moment, the requested data can be gathered into the network buffer through the view from the collective buffer *b* (line 18). The gathered data are then sent to the MPI process (line 20). Finally, if the prefetching buffer pool is not full, it is filled by predicting future block accesses based on the locally stored views (line 22) and scheduling them for prefetching (line 23). Future access prediction in line 22 looks at blocks accessed by the current operation (*MAPPED_FILE_BLOCK_SET*) and predicts potential future access blocks based on the current views of all processes (*view_table*).

The I/O thread at the aggregator runs Algorithm 3. All blocks that have been scheduled for prefetching (line 1) and whose prefetching has finished (line 23) are moved to the collective buffer cache (lines 10-11). If the collective buffer pool is full, block replacement is performed (lines 2-10).

Algorithm 3 Prefetching: Aggregator-side

```
1: MAPPED_FILE_BLOCK_SET ← map(view_table[mpi_rank], left_view_offset,
   right_view_offset)
2: for all b in MAPPED_FILE_BLOCK_SET do
3:   if b not in CollBufferPool then
4:     if full(CollBufferPool) then
5:       repl_b ← replacementBlock(CollBufferPool)
6:       if dirty(repl_b) then
7:         write(repl_b)
8:       end if
9:       remove(repl_b, CollBufferPool)
10:    end if
11:    insert(b, CollBufferPool)
12:    if b in PrefetchBufferPool then
13:      remove(b, PrefetchBufferPool)
14:    else
15:      read(b)
16:    end if
17:  end if
18:  gather(netbuf, b, left_view_offset, right_view_offset,view)
19: end for
20: send(netbuf, mpi_rank)
21: if not full(PrefetchBufferPool) then
22:   PREFETCH_BLOCK_SCHEDULE ← predict_access(view_table, MAPPED_
      FILE_BLOCK_SET)
23:   insert(PREFETCH_BLOCK_SCHEDULE, PrefetchBufferPool)
24:   signal_io_thread()
25: end if
```

3.3.4 File access semantics and consistency

Traditional solutions for both overlapping I/O problems use whole file or byte-range file locking to ensure exclusive access to the overlapping regions and bypass the file system cache [PTH+01, LD02]. Unfortunately, not only can file locking serialize I/O, but it can also increase the aggregate communication overhead between compute and I/O nodes. For solving the file consistency problem across multiple I/O operations, in [Wei06], the authors propose a method called *Persistent File Domains*, which tackles client-side cache coherency with additional information and coordination to guarantee safe cache access without using file locks.

In our generic parallel I/O architecture, data consistency is enforced in the hierarchy by not allowing more than one copy of a file block at any cache level (the file blocks have the same size at both client-side and I/O node-side levels). In the client-side file cache management tier each file block is managed (cached, modified, written-back, prefetched) by exactly one aggregator. Each aggregator is mapped on exactly one I/O node. Therefore, the file block is managed by exactly one I/O node-side file cache management module. This decision is motivated by the frequent access patterns of the parallel applications: individual processes write non-overlapping file regions and there is a high inter-process spatial locality. Data are transferred between cache

Algorithm 4 Prefetching: I/O thread-side

```
1: for all b in PrefetchBufferPool do
2:   if full(CollBufferPool) then
3:     repl_b ← replacementBlock(CollBufferPool)
4:     if dirty(repl_b) then
5:       write(repl_b)
6:     end if
7:     remove(repl_b, CollBufferPool)
8:   end if
9:   read(b)
10:  insert(b, CollBufferPool)
11:  copyBlock(b, PrefetchBufferPool, CollBufferPool)
12:  remove(b, PrefetchBufferPool)
13: end for
```

levels always at block granularity. For writing to a file block less than its size in the client-side or I/O node-side caches, a read-modify-write operation is needed.

Our solution provides a relaxed file system semantics, motivated by the well known fact that POSIX-like semantics are not suitable for HPC workloads [hs95]. While a file is open, its data may reside any level of the cache hierarchy. Data is ensured to have reached the final storage after file close or after a file sync has been executed. In particular, MPI provides three levels of consistency: sequential consistency among all accesses using a single file handle, sequential consistency among all accesses using file handles created from a single collective open with atomic mode enabled, and user-imposed consistency among accesses other than the above. The atomic mode for independent I/O file accesses has not been implemented, i.e. sequential consistency is not guaranteed for concurrent independent I/O with overlapped access regions. This approach is similar to the one taken in PVFS and it is motivated by the fact that overlapping accesses are not frequent for parallel applications. Nevertheless, the atomic mode can be enforced as a user defined consistency semantics by using `MPI_FILE_SYNC` as described in the MPI standard [Mes95].

3.4 Summary

We have proposed a generic parallel I/O architecture based on six layers: application, application I/O forwarding, client-side file cache, aggregator I/O forwarding, I/O node-side file cache, and storage system. In order to hide the latency of file accesses, the architecture contains two coordinated cache modules on compute nodes and I/O nodes. The design of both file cache management modules targeted the separation of mechanisms from policies and decoupling the transfers between layers and the functionality provided at each layer. Each of these caches is managed by a dedicated data staging thread, which asynchronously stages the data between the local cache and the next level in the hierarchy. Subsequently, the data is asynchronously transferred to/from the I/O nodes or storage nodes. Additionally, we have described the common components of our architecture for both clusters and supercomputers, namely generic client-side file cache management. In the following chapters we describe the specific design for cluster and supercomputer systems.

Chapter 4

Parallel I/O architecture for clusters

The majority of existing approaches apply mostly to cluster architectures. This is due to the fact that supercomputers were to a large extent proprietary, limiting the research opportunities. Large-scale computing clusters with hundreds to tens of thousands of processors are being increasingly used to execute large, data-intensive applications in several scientific domains. Such domains include, for example, high-resolution simulation of natural phenomenon, large-scale image analysis, climate modelling, and complex financial modelling. The I/O requirements of such applications can be staggering, ranging from terabytes to petabytes and beyond, and managing such massive data sets has become a significant bottleneck in application performance [DL08]. Thus solving this I/O scalability problem has become a critical challenge in high-performance computing.

In this chapter we describe how the generic parallel I/O architecture can be applied to clusters of off-the-shelf components. We present the design for two possibilities of I/O architectures on clusters.

4.1 Architecture overview

In this subsection we describe how the generic parallel I/O architecture from the previous chapter can be applied to clusters of off-the-shelf components.

For cluster I/O systems we propose two possible scenarios. For file-system independent parallel I/O architectures or in which no parallel file system is running, we propose the Ad-Hoc Parallel I/O System (AHPIOS). On the other hand, for deployed parallel I/O architectures, we show how popular file systems, like GPFS, can take advantage from our generic parallel I/O architecture.

The parallel I/O architecture for clusters maps to our generic architecture in the following way (as shown in Figure 4.1). On the left of the figure, we represent how the generic architecture maps for file-system independent parallel I/O architectures, using AHPIOS. On the right, for deployed parallel I/O architectures, we show how popular file systems such as GPFS, can take

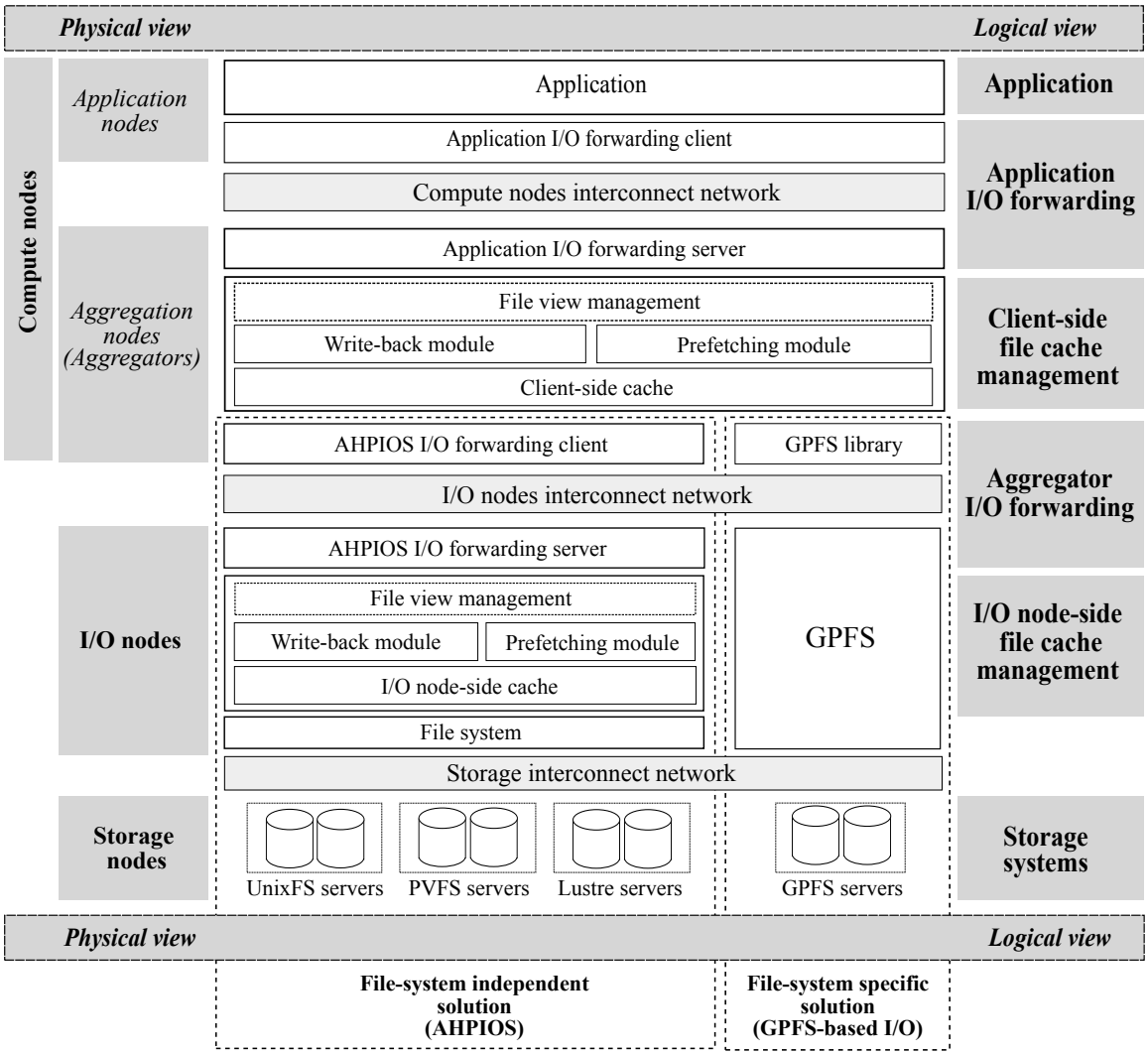

Figure 4.1: *General parallel I/O architecture for clusters. Client-side and I/O node-side file view management modules are optional.*

advantage from our generic parallel I/O architecture. Application, application I/O forwarding, and the client-side file cache management modules are common for both deployed parallel I/O solutions. Dotted boxes represent that file view management modules are optional on both levels.

Applications issue file access calls through file interface, such as FUSE or MPI-IO. Application I/O forwarding module transfers data between applications and client-side file cache module. Application I/O forwarding module handles sequential and parallel file access which is forwarded using the MPI-IO interface. The client-side file cache module is common for both file-system independent and dependent solutions.

Aggregator I/O forwarding and I/O node-side file cache modules are different for each file system solution. For file-system independent platforms, the aggregator I/O forwarding and I/O node-side file cache modules are based on AHPIOS. More details about AHPIOS I/O forwarding mechanism are given in Section 4.2.2. On the other hand, for GPFS I/O architectures, the aggregator I/O forwarding is handled directly by the file system. In this case, the aggregator I/O

forwarding module takes advantage of specific GPFS parameters in order to increase file access performance.

4.2 File-system independent solution

File systems, such as Sorrento [TGZ+04] and RADOS [WLBM07] (RADOS is a part of Ceph scalable high-performance distributed file system [WBM+06]), offer scalable auto-reconfigurable storage solutions for dynamic pools of storage resources. The physical placement of logical data segments in these systems is hidden from the applications. These systems do not target the optimal mapping of parallel applications access patterns on the storage layout. Additionally, data access optimizations such as view-based I/O (presented in Section 3.3.2) and independent operations can not be used on top of these storage systems.

To overcome with these problems, we propose AHPIOS, a light-weight Ad-Hoc Parallel I/O System that can be used as a middleware located between MPI-IO and distributed storage resources, providing high-performance scalable access to files. AHPIOS can be used as a light-weight low-cost alternative to any parallel file system, virtualizing on-demand independent storage resources.

4.2.1 Architecture overview

AHPIOS can be mapped to our generic parallel I/O architecture from Figure 3.1, in the following way. AHPIOS architecture is as well organized on six tiers: application, application I/O forwarding, client-side cache, AHPIOS I/O forwarding, AHPIOS I/O node-side cache and storage. Applications issue file access calls through the MPI-IO interface. Application I/O forwarding layer transfers data between applications and client-side cache module through the compute nodes interconnect network. Client-side file cache management tier manages a cache on compute nodes, offers access to the applications (optionally through views), and transfers data between compute nodes and AHPIOS servers over the I/O network. AHPIOS I/O forwarding tier transfers data between client-side cache and I/O node-side cache modules through the I/O nodes network. I/O node-side file cache management tier manages a cache on I/O node, serves requests from the client-side file cache management tier, allows the virtualization of file system, and accesses file systems over the storage network.

The architecture supports different network configurations, offering flexibility in the deployment. The proposed architecture is not only oriented to cluster systems with multiple interconnection networks. For example I/O nodes can be deployed in the same physical nodes where application nodes are deployed, sharing the interconnection network for computation and I/O.

As illustrated in Figure 4.2, the AHPIOS client-side layer is integrated into the ROMIO architecture stack by deploying the C4 and C5 layers. The C4 layer can be divided into two sublayers: C4.1 and C4.2. C4.1 maps the ADIO file operations onto I/O tasks to be performed by the individual AHPIOS servers. These can be metadata-related, such as creating or deleting a file or data operations. In C4.2, the I/O tasks are scheduled for transfer by a parallel I/O scheduling module [ISCG06]. AHPIOS I/O forwarding client resides in C5 layer, and is responsible for communication with the AHPIOS servers.

The AHPIOS servers manage the I/O forwarding and I/O-node side file cache modules The AHPIOS servers run as a completely client-independent MPI application. As shown in

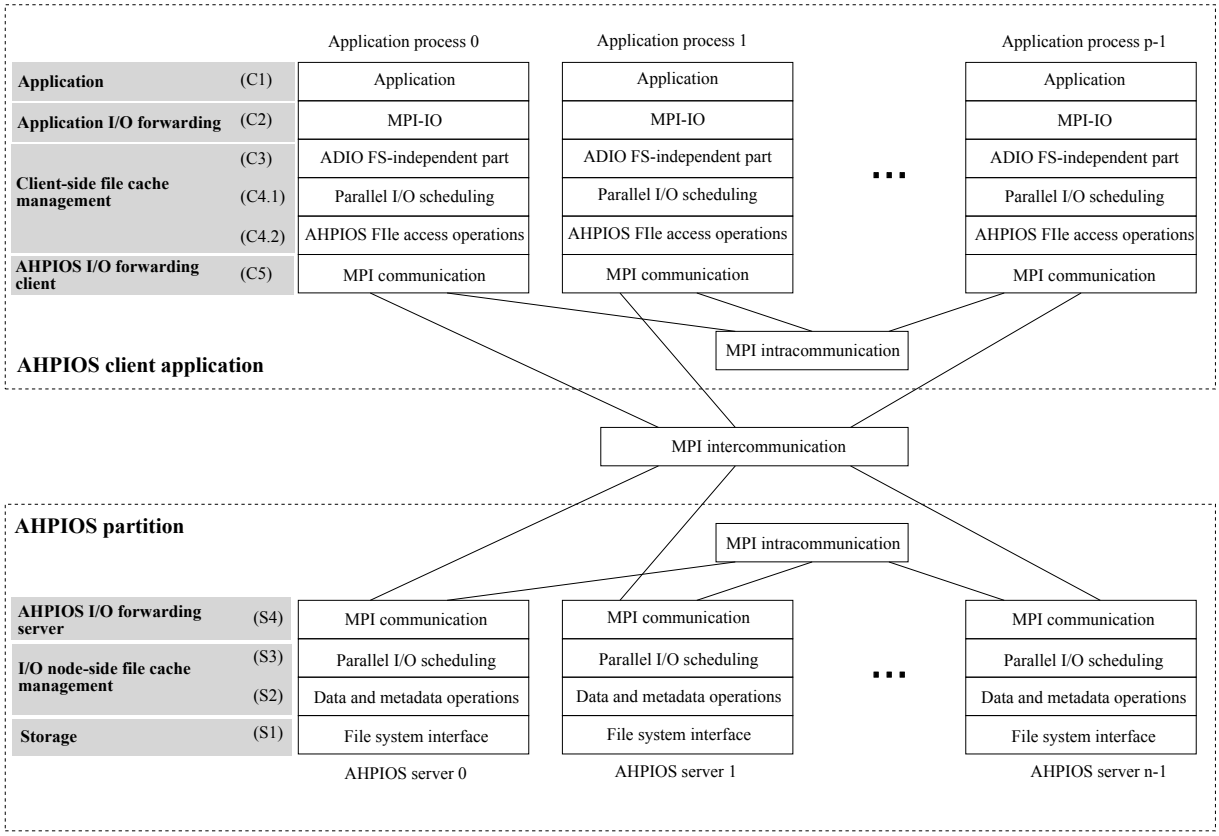

Figure 4.2: *AHPIOS software architecture with one application and one AHPIOS partition.*

Figure 4.2, the server design is structured in four sublayers. Communication with the client is performed through MPI routines in the S4 sublayer through the AHPIOS I/O forwarding server. The S3 sublayer is responsible for the parallel I/O scheduling policy, which cooperates with the corresponding modules at the client side. Data and metadata management is performed in the S2 sublayer. Finally, the S1 layer transfers data and metadata to the final storage system.

AHPIOS manages two levels of caching as depicted in Figure 4.1. First, client applications cache collective buffers at aggregator nodes, corresponding with the generic client-side file cache management tier. Second, AHPIOS servers perform data caching by collectively managing a single-copy cache at I/O nodes (I/O-node side file cache management). In AHPIOS, as in the client-side file cache module, the collective buffers are cached at the AHPIOS servers across collective and not collective I/O operations.

An AHPIOS partition is managed by a set of AHPIOS servers, which are also processes of an MPI program, running independently from the MPI application. Figure 4.2 shows the software architecture of the AHPIOS system: a client application in the upper part, and AHPIOS servers in the lower part.

4.2.2 Design and implementation

Is this section we extend some design and implementation considerations of AHPIOS.

AHPIOS is completely implemented using the Message Passing Interface (MPI). MPI brings the advantages of portability, scalability and high-performance. AHPIOS allows applications to dynamically manage and scale distributed partitions in a convenient way.

The system manages a hierarchy of distributed file caches as depicted in Figure 4.1. First, client applications cache collective buffers on the client-side file cache module. Second, the AHPIOS servers also perform data caching by collectively managing a single-copy cache on the I/O-node side file cache module. The communication within and between these layers is performed through standard MPI communication operations.

A data staging strategy hides the latency of transferring data blocks between the levels of the cache hierarchy. The data transfer between the client-side collective cache and AHPIOS server caches, and between AHPIOS server caches and disk storage is done asynchronously. Therefore, overlapping of computation, communication and I/O is achieved.

Given an MPI application accessing files through the MPI-IO interface and a set of distributed storage resources, AHPIOS constructs on demand a distributed partition, which can be accessed transparently and efficiently. On each AHPIOS partition the users can create a directory name space in the same way as on any regular file system. Files stored on one AHPIOS partition are transparently striped over storage resources as in any parallel file system. Each partition is managed by a set of storage servers, running together as an independent MPI application. The access to an AHPIOS partition is performed through an MPI-IO interface. A partition can be built, scaled up and down on demand during the application run-time. Multiple AHPIOS partitions can coexist, each being managed by different sets of AHPIOS servers with different configurations.

The MPI-IO mechanisms and optimizations, such as file view setting and collective I/O, are strongly integrated into AHPIOS storage system. MPI views may be set either at client- or I/O-node sides, and MPI collective buffering, the mechanism behind two-phase I/O can be activated either at clients (close to computing) or at the I/O servers (close to storage system layer).

AHPIOS is started by parsing a configuration file, which defines the parameters, such as number of storage resources, file stripe size, type of collective I/O to be used, sizes of client and server caches, type of parallel I/O scheduling policy, etc. The user can control both the MPI-IO and parallel I/O system through the configuration file.

The partition creation attributes such as the number of resources, the stripe size and the list of resources used for data and metadata storage are specified in a configuration file, as in the example shown below:

```
# The default stripe size of the partition
stripe_size = 64k
# Number of IOS
nr_ios = 4
# Storage resources of AHPIOS servers
ahpios_server = n0:/data
ahpios_server = n1:/data
ahpios_server = n2:/data
ahpios_server = n3:/data
# Path of the metadata directory
metadata = n0:/data
```

An AHPIOS partition is created by the first application that uses the partition. The storage servers are spawned through the MPI dynamic process mechanism and the partition is subsequently registered in the global registry identified by a port name. Subsequently, other applications can mount the partitions identified by the port name and employing the client/server functionality of MPI2.

Several AHPIOS partitions with different configurations may be running in parallel on a cluster of computers, after registering with the *global registry* (more details in Section 4.2.2). Figure 4.3 shows an example of two applications sharing two different AHPIOS partitions. For each mounted partition, a dedicated MPI intercommunicator is created, through which the MPI-IO client layer communicates with the AHPIOS servers.

Finally, AHPIOS can be started on-demand on a computing system with distributed storage resources. The approach is useful in several scenarios. First, it can be employed as a middle layer between MPI applications and existing I/O servers. The configuration can be done on demand, at application start-up, allowing for the tailoring of the system parameters to the application needs. Second, a system with dynamical availability of resources may employ AHPIOS for storage virtualization on-the-fly. Third, different parallel file systems can be virtualized in order to increase the parallelism degree. Fourth, it allows the virtualization and the parallel use of different storage resources in a distributed system, in which no parallel file system is running.

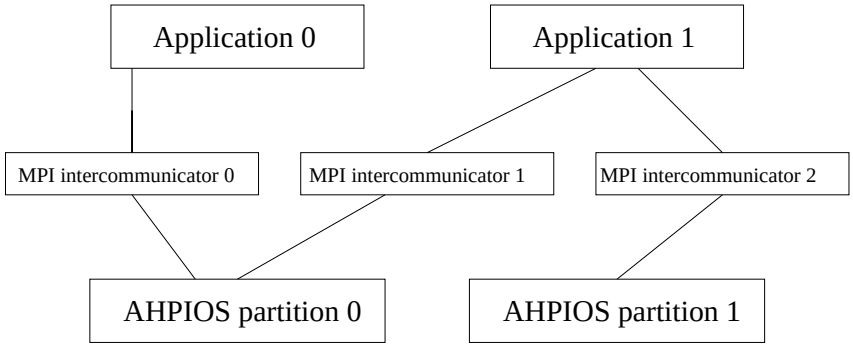

Figure 4.3: *AHPIOS partitions accessed by two MPI applications.*

Interconnection

The AHPIOS I/O forwarding module is based on MPI intracommunicators and intercommunicators (as shown in Figure 4.2).

The AHPIOS servers are interconnected through an *MPI intracommunicator*. An MPI intracommunicator is an MPI mechanism that allows the members of a group of processes to communicate among each other through MPI communication routines. The aggregators communicate among each other through an intracommunicator, too.

The AHPIOS I/O forwarding clients communicate with AHPIOS the I/O forwarding servers through an *MPI intercommunicator*. An MPI intercommunicator is an MPI mechanism that allows the members of different process groups to communicate. Two different applications communicate also through an MPI intercommunicator.

Elastic partitions

AHPIOS partitions are elastic: they can scale up and down by increasing the number of storage resources. Scaling up involves only a system restart with a larger number of AHPIOS servers. In this case, for files that are stored on the old smaller set of storage resources, users can choose either to preserve their initial structure or to redistribute them over the newly available storage resources. Mounting a scaled down partition involves a redistribution of file data from the storage resources that become unavailable to the remaining ones. This is a two-step process. First, all the old storage resources are mounted by starting the system with the old number of AHPIOS servers and the redistribution is performed. Second, the old partition is unmounted and the new partition is mounted.

File cache management

Data access is performed through the cooperation of the clients running on several compute nodes and the AHPIOS servers. Data transfer order is controlled by a parallel I/O scheduling strategy, which is described in [ISCG06].

An AHPIOS file may be striped over several AHPIOS servers. By default the files are striped over all the available AHPIOS servers, but the user can control the striping parameters through MPI hints.

An AHPIOS partition is accessed through a two-level hierarchy of distributed file caches as described shortly in the Section 3.1 and shown in Figure 4.4.

The first level of caching corresponds with the generic client-side file caching (presented in Section 3.3.1). The first distributed caching level works in the following way. File blocks are mapped in a round-robin fashion over all aggregators. The file requests are directed accordingly to the responsible aggregator. The aggregator clusters together several requests before accessing the next level of caching. Communication with the second level of caching is performed asynchronously by an I/O thread, which hides the file access latency from the application.

The AHPIOS servers manage the second level of distributed file caching at I/O nodes, which is specific for file-system independent systems. File blocks are mapped to AHPIOS servers in a round-robin fashion and each server is responsible for transferring its blocks to and from the persistent storage. When an AHPIOS server receives a request for a block assigned to another server, it serves this request in cooperation with the other servers. This approach is useful at least in two scenarios. First, the I/O related computations can be offloaded to the AHPIOS servers. Second, a group of application processes is assigned to an I/O server. The I/O server is responsible for all file requests from this group, and can serve them in cooperation with other I/O servers, similar to the I/O system of the IBM Blue Gene supercomputers.

The user can choose to disable the first level of distributed file cache for consistency reasons, when multiple client applications share the same files. However, studies have shown that this is rarely the case with scientific applications [SR97, WXH+04].

Data access operations

As discussed in Section 2.3, the MPI-IO standard defines two major groups of file access operations: *collective* and *independent*. In AHPIOS, the independent operations are identical with the server-directed I/O operations; in the sense that the data is transferred between MPI pro-

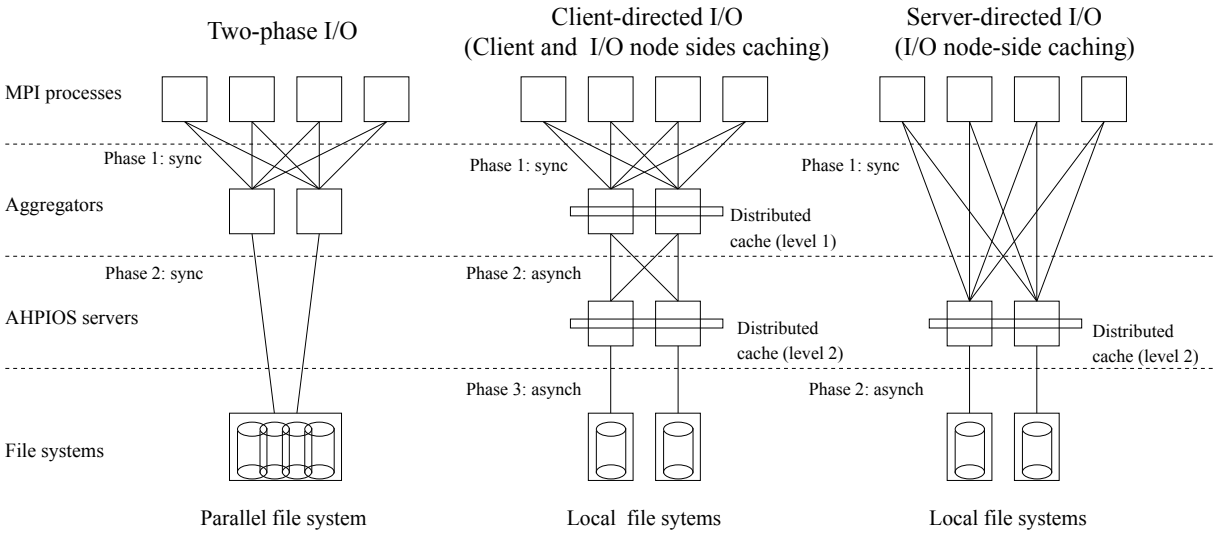

Figure 4.4: *Comparison of data flow in two-phase I/O, client-directed I/O and server-directed I/O.*

cesses and AHPIOS servers and the AHPIOS servers merge independent small requests into larger collective requests and cache the data in collective buffers. Therefore, the independent I/O operations can perform as efficiently as collective I/O operations.

Collective operations are suitable for parallel workloads because of four basic characteristics that are common in the data-intensive parallel scientific applications [NKP+96, SR97, SR98, CACR95, WdW03, FOR+00]. First, in many cases, all processes of one parallel scientific application perform shared access to the same file. Second, each individual compute node accesses the file non-contiguously and with small granularities. Third, there is a high degree of spatial locality: when a node accesses some file regions, the other nodes tend to access neighbouring data. Fourth, write accesses are mostly non-overlapping among processes.

File view management modules

ROMIO adopts the two-phase I/O strategy, for which the collective buffers only reside at the aggregators. AHPIOS includes two collective I/O methods,both of them based on views: *view-based client-directed* and *view-based server-directed*. Both of these methods are based on generic view-based I/O method presented in Chapter 3. Each of them has its own benefit, as we will demonstrate in the experimental section.

Figure 4.4 depicts two-phase I/O, client-directed I/O, and server-directed I/O. We assume that the view has already been declared. After the view declaration is made, the view description is kept at client processes in two-phase I/O. For client-directed I/O the view is sent to aggregators, where it is stored for future use. For server-directed I/O it is transferred to the AHPIOS servers, which also store it for future use.

Storing the view remotely has the potential of significantly reducing the overhead of transmitting non-contiguous file regions, which map contiguously to data locally available at the MPI processes. First, this contiguous data do not have to be processed locally (scattered or gathered), and no offset-length lists have to be sent for each access (as in the case of two-phase I/O). Second, the view is sent only once in a compact form and can be reused, when the same access pattern

appears repeatedly (a frequent behaviour of parallel applications).

For both two-phase I/O and client-directed I/O, the aggregators represent a subset of the MPI processes, and are used for merging small requests into larger ones. By default, all MPI processes act as aggregators. However, the user may set the number of aggregators by an MPI hint.

Two-phase I/O consists of two synchronous phases, which correspond to shuffle phase (1) and I/O phase (2) described in Figure 4.4.

Client-directed I/O consists of three phases. When client-directed I/O starts, the views have been already stored at the aggregators at view declaration. The first phase (1) is synchronous and consists of shuffling the data between MPI processes and AHPIOS aggregators: small file regions are gathered at aggregators for writing and are scattered from aggregators for reading. The small file regions are transferred contiguously between each pair of MPI processes and aggregators and are scattered/gathered by using the view previously stored at aggregator at view declaration. In the second phase (2) data are asynchronously staged from first level distributed file cache of aggregators to the AHPIOS servers. Only full file blocks are transferred between these two cache levels. Finally, in the third phase (3) data are also asynchronously staged from the second level distributed file cache of AHPIOS servers to the final storage.

Server-directed I/O consists of two phases. When server-directed I/O starts, the views have also been already stored at the AHPIOS servers. The first phase (1) is synchronous and similar with the client-directed I/O, excepting the fact that the data are shuffled between MPI processes and AHPIOS servers. The second phase (2) is asynchronous and is the same as the third phase in the client-directed I/O.

It can be noticed that there are two main differences between client-directed I/O and server-directed I/O: the place where the view is stored and the intermediary level of caching for the client-directed I/O. Consequently, small requests are merged into larger ones at aggregators for client-directed I/O and at AHPIOS servers for server-directed I/O.

Table 4.1 compares two-phase I/O with client-directed and server-directed I/O. Unlike in two-phase I/O, for client-directed and server-directed I/O, the views, represented as MPI data types, are not stored at the client application, but decoded at the MPI-IO layer, serialized and transferred either to AHPIOS aggregators or AHPIOS servers. Upon receiving the view, the aggregators or servers unserialize and reconstruct the original view data type. The advantage of this approach is that no metadata has to be sent over the network at access time, because the view representing the file access pattern is already stored remotely. For two-phase I/O the access pattern generated by the view must be sent as lists of (file offset, length) tuples. In the case of client-directed and server-directed I/O the data can be transferred contiguously between client and aggregator/server.

In two-phase I/O, data are not cached at aggregators, but only temporarily buffered and synchronously transferred to the file system. In client-directed I/O the file data may be cached at both levels of the distributed file caching hierarchy and they are transferred asynchronously to the final storage. In server-directed I/O, the data are cached in the second level distributed file caching hierarchy and they are asynchronously sent to the storage. The asynchronous transfers allow a transparent overlapping of computing, I/O related communication and storage access.

Table 4.1: *Comparison of three collective I/O methods.*

Operation	Two-phase I/O (2PIO)	Client-directed (VBIO-CS-IONS)	Server-directed (VBIO-IONS)
View declaration	Store view at client	Send view to AHPIOS aggregator	Send view to AHPIOS server
File Access (metadata)	Generate file-offset lists from views and send them to TP aggregators	No action	No action
File Access (at client)	Non-contiguous	Contiguous access	Contiguous access
File Access (aggregator)	Non-contiguous	Non-contiguous	n/a
File Access (file server)	Contiguous	Contiguous	Non-contiguous
File cache	No	In two levels of distributed cache	In second level of distributed cache
Data staging from aggregators to servers/FS	Synchronous	Asynchronous	n/a
Data staging from servers to storage	n/a	Asynchronous	Asynchronous
File close	No action	Ensure AHPIOS aggregators and servers flush all data	Ensure AHPIOS servers flush all data

Cache coherency and file consistency

The cache coherency and file consistency mechanisms were presented in Section 3.3.4. In this section we explain how our generic cache coherency mechanisms are applied in AHPIOS.

For client-directed I/O, the modifications of single-copy file blocks are always performed in the first level cache by one aggregator. Server-directed I/O does not use the first level cache, therefore, the coherency is enforced only at servers by allowing one copy of a file block. A server-directed I/O operation triggers the eviction of accessed file blocks from the first level cache to the server, before performing its own operations.

According to MPI standard, MPI provides three levels of consistency: sequential consistency among all accesses using a single file handle, sequential consistency among all accesses using file handles created from a single collective open with atomic mode enabled, and user-imposed consistency among accesses other than the above.

AHPIOS provides partial MPI consistency. Collective accesses from one application are sequentially consistent. The data are flushed to the end storage for both cache levels either upon calling `MPI_FILE_SYNC` or closing the file. The atomic mode for independent I/O file accesses has not been implemented, given that overlapping accesses are not frequent for parallel applications, and that it can be enforced as a user defined consistency semantics by using `MPI_FILE_SYNC` as described in the MPI standard.

Metadata management

In AHPIOS there are two levels of metadata management: global metadata management of AHPIOS partitions and local metadata management of individual AHPIOS partitions.

Global metadata management In AHPIOS there is no server that performs global metadata management. The global metadata are minimal, and contain only information about the particular partitions of the file system. This information is stored in a file shared by all partitions and called *registry*. Each set of AHPIOS servers managing a partition accesses atomically this file in order to read or modify it. This access should not cause a bottleneck in a large system, because the registry is accessed only when a partition is created, mounted or unmounted. All these operations are infrequent.

The global registry stores structural and dynamic configuration parameters of the AHPIOS partition. The structural parameters are partition name, storage resources assigned to the AHPIOS servers, number of AHPIOS servers, default stripe size and metadata file disk location. The dynamic parameters include network buffer size, parallel I/O scheduling policy, buffer cache size of the AHPIOS server etc. The dynamic parameters can be changed by the user each time a partition is mounted.

Local metadata management For each partition, one of the AHPIOS servers plays also the role of a partition-local metadata manager. This server manages a local name space and an inode list, stores and retrieves the file metadata and updates the metadata in coordination with other servers. The global name of a file is given by appending the local path of a file to the global unique partition name. The local name space can be as simple as a directory in the name space of the local file system on the node where the AHPIOS metadata server is running. An inode stores typical file metadata, including values for stripe size and number of stripes. By default, the number of stripes is the same as the number of AHPIOS servers and the user can modify this value through a hint.

4.3 File-system specific solution

This section presents how the generic parallel I/O architecture can be mapped with a specific file system, in this case GPFS.

GPFS is the IBM's parallel file system solution for cluster and supercomputers, and has been introduced in Section 2.2. A previous work [PTH+01] has presented an MPI-IO implementation for GPFS inside the IBM MPI. This implementation was proprietary and, to the best of our knowledge, has never been released to the public domain. The GPFS-based parallel I/O architecture targets to fill this gap and, additionally, to enhance the utilization of the storage and network resources in GPFS-based architectures. Yu et al. [SHea06] present a GPFS-based three-tiered architecture for Blue Gene/L. The tiers are represented by I/O nodes (GPFS clients), network-shared disks, and a storage area network.

Our solution focuses on extend this hierarchy to include the memory of the compute nodes and proposes an asynchronous data-staging strategy that hides the latency of file accesses from the compute nodes.

4.3.1 Preliminaries

GPFS is highly optimized for large-chunk I/O operations with regular access patterns (contiguous or regularly strided). However, its performance for small-chunk, non-contiguous I/O operations

with irregular access patterns (e.g. non-constant strides) is not sufficiently addressed. This kind of access can cause a high access contention, translated especially in a high locking overhead.

GPFS addresses this issue by a technique called data-shipping [SH02, PTH+01], which can be activated/deactivated through a hint of the GPFS library. This technique disables *client-side* caching and binds each GPFS file block to a single I/O agent, which will be responsible for all accesses to this block. For write operations, each task sends the data to be written to the responsible I/O agents. I/O agents in turn issue the write calls to the end storage system. For reads, the I/O agents read the file blocks, and ship only the requested read data to the appropriate tasks. This approach is similar to the two-phase I/O, described in Section 2.5.2. Data shipping is more efficient than the default locking approach, when fine-grained sharing is present, because the granularity of GPFS cache consistency is an entire file block, and accesses to the same block are serialized by the locking manager.

GPFS recognizes sequential and simple strided file access patterns for read and write operations [AR] and performs prefetching and write-behind accordingly. However, for different patterns, the default prefetching and write-behind policy may become non-productive. For these cases, the user can define customized prefetching and write-behind policies by a hint called Multiple Access Range (MAR) [PTH+00]. The MAR hint allows each process to specify at file system block granularities the file regions over which prefetching and write-behind should be performed. Subsequently, GPFS applies these policies accordingly. There are system limits on the amount of buffer cache used for write-behind and prefetching, as it will be demonstrated by the experimental measurements.

The performance of GPFS is well tuned for large accesses at block granularities. Therefore, we mainly target the optimization of both contiguous and non-contiguous file accesses with small and medium granularities, such as the one exhibited by a significant class of scientific applications [NKP+96, SR97].

4.3.2 Architecture overview

In this subsection we present how the generic parallel I/O architecture is mapped with GPFS file system, namely GPFS-based parallel I/O.

Our GPFS-based parallel I/O architecture can be mapped to our generic parallel I/O architecture from Figure 3.1, in the following way. GPFS-based parallel I/O architecture architecture is organized on four tiers: application, application I/O forwarding, client-side cache, and storage. Applications issue file access calls through the MPI-IO interface. Application I/O forwarding layer transfers data between applications and client-side cache module through the compute node interconnect network. Client-side file cache management tier manages a cache on compute nodes, offers access to the applications (optionally through views), and transfers data between compute nodes and GPFS servers. Finally, the storage system consists of file system servers running on storage nodes and accessing disks over a NSD or SAN attached GPFS storage subsystem.

In the client-side file cache management module, the file access latency is hidden from the applications through an asynchronous data staging strategy at aggregator nodes. The client-side file cache management module used correspond with the generic client-side file cache tier. However, in order to take advantage of specific GPFS optimizations, the prefetching module needs to be modified, as shown in Section 4.3.3. The I/O node-side file cache module and the storage system are handled by the GPFS infraestructure. In this case, our contribution is available only

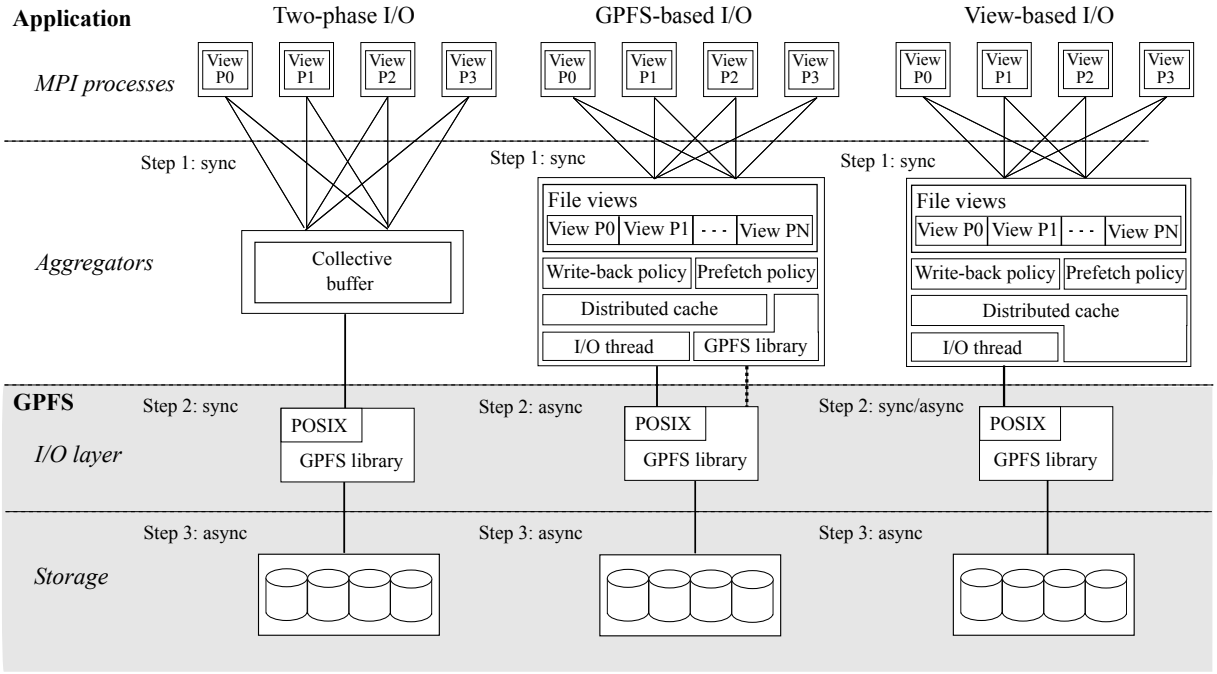

Figure 4.5: *Comparison of two-phase I/O, view-based I/O, and GPFS-based I/O. The dotted line represents file system communications through GPFS library.*

on the client-side file cache tier due to two factors. First, GPFS is a proprietary storage solution and the source code is not available. Second, GPFS file system includes an asynchronous method in the I/O nodes, on which our solution is partially based.

4.3.3 Design and implementation

MPI-IO calls of the applications are translated in MPI-IO into a smaller subset of ADIO calls. MPI applications issue file access calls through MPI-IO interface. Application I/O forwarding layer transfers data between applications and *client-side cache* module through MPI. In the *client-side* file cache management module, the file access latency is hidden from the MPI applications through an asynchronous data staging strategy at aggregator nodes.

The client-side file cache module is integrated into the ROMIO architecture stack, using the ADIO layer. The ADIO layer can be divided into two sublayers: ADIO-FS dependent and independent. ADIO-FS dependent maps the ADIO file operations onto I/O tasks to be performed by the individual I/O servers at I/O nodes. The I/O requests of the application are sent by the aggregators, which will then forward them to the I/O servers through the GPFS library.

GPFS-based I/O is depicted in the middle part of Figure 4.5. The file write accesses are scattered from the MPI processes directly to the GPFS through POSIX calls. Because the local GPFS buffer caches are disabled, the data are always transferred to the GPFS agents. Subsequently, the agents asynchronously write the data to the final storage. The file read operation is based on the prefetching mechanism of GPFS and on MPI views.

Data access operation

A collective file write strategy based on GPFS data-shipping, and a view-based collective I/O mechanism, relying on GPFS mechanisms, are at the core of the novel optimizations proposed. Another principal components of the GPFS-based architecture are the write-back and prefetching modules, which have been integrated in ROMIO.

View-based collective I/O includes a thread-based flushing method implementing a write-back policy for latency hiding, and a prefetching method, based on GPFS hints, to increase small read access performance.

Data shipping-based I/O. Our write GPFS-based solution relies on a collective file write strategy based on GPFS data-shipping, namely data shipping-based I/O (DSIO). DSIO was done in the file system dependent layer from Figure 2.11. The data-shipping mode is activated for a given file in GPFS through a hint offered by the GPFS library *gpfsDataShipStart*). Subsequently, GPFS uniquely assigns each file block to one I/O agent (by default round-robin), and all file accesses for the assigned block go through this agent. Compute node file caching and file locking is disabled. Performing block modification only at the agent avoids trashing and locking overhead at the cost of lower access locality.

The user can control other parameters of data-shipping through hints: the number of I/O agents, the file block assignment to I/O agents, and the sizes of file blocks.

MAR prefeching-based I/O. Multiple Access Range (MAR) is a GPFS hint and defines the customized user prefetching and write-behind policies, and has been introduced in Section 4.3.1.

The GPFS file read operation method is based on the prefetching mechanism of GPFS and on MPI views, namely MAR prefetching-based I/O (MARPIO). GPFS manages a prefetch buffer pool of a few buffers with the size of a file system block. The prefetching policy is enforced at the aggregators and consists of: (1) deciding which blocks to prefetch (prediction), (2) monitoring the block prefetching, and (3) moving the prefetched blocks from prefetch buffer pool to the collective buffer pool. Prefetching is based on MPI views, which reflect potential future access patterns. As described in the previous section the views are stored at aggregators at view declaration.

At access time, a compute node sends to all aggregators only the boundaries of the file access interval of the view: left view offset (*left_view_offset*) and right view offset(*right_view_offset*) (step 1 of Figure 4.5).

Subsequently, each aggregator runs Algorithm 5 for each compute node read request. It is important to note that lines from 1 to 23 of the algorithm correspond with of generic *client-side* solution for prefetching. First, the set of blocks to be read from, (*MAPPED_FILE_BLOCK_SET*) is calculated by mapping the view access interval onto file blocks through the previously stored view (line 1). For each block b from *MAPPED_FILE_BLOCK_SET* (line 2), if the block is not in the collective buffer pool (line 3), it is inserted (line 11), after replacing a block if the collective buffer pool was full (lines 4-10). If has been already read into GPFS prefetching pool (line 12), i.e. has been scheduled for prefetch previously, the content of b is copied to the collective buffer pool (line 13) and b is removed from prefetching buffer (line 14). If b has not been scheduled for prefetching, it is read directly from GPFS (line 16). At this moment, the requested data can be gathered into the network buffer through the view from the collective buffer b (line 19). The gathered data are then sent to the MPI process (line 21). All blocks that have been scheduled for

prefetching (line 22) and whose prefetching has finished (line 23) are moved to collective buffer cache (lines 32-33). If the collective buffer pool is full, block replacement is performed (lines 24-31). Finally, if the prefetching buffer pool is not full, it is filled by predicting future block accesses based on the locally stored views (line 37) and scheduling them for prefetching (line 38).

We note that, in the case of reading, prefetching is performed asynchronously inside GPFS, which also keeps track of the *PrefetchBufferPool*. *PrefetchBufferPool* is made available to the user in the form of an array of structures upon the activation of customized prefetching through the MAR hint. Subsequently, our approach can check the termination of a prefetching operation in line 23 by monitoring the *PrefetchBufferPool*.

Future access prediction in line 37 looks at blocks accessed by the current operation and predicts potential future access blocks based on the current views of all processes (*view_table*).

4.4 Summary

In this chapter, we have presented our generic parallel I/O architecture for clusters.

We have presented the AHPIOS parallel I/O system, an ad-hoc parallel system that allows on-demand virtualization of distributed resources, provides high performance parallel I/O and can be used as cost-efficient alternative to the traditional parallel file systems. AHPIOS is completely implemented in MPI and offers a scalable efficient platform for parallel I/O. The two-level distributed file cache scales with the number of processors at the first level and with the number of storage resources at the second level. The strategy of asynchronous data staging between the caching levels hides the latency of file accesses from the applications.

Finally, we have described the design and implementation of our GPFS-based parallel I/O architecture, including the design and integration of GPFS-based write-back and prefetching modules into view-based I/O. Our solution relies on a collective file write strategy based on GPFS data-shipping, and a view-based collective I/O mechanism including a thread-based flushing method implementing a write-back policy for latency hiding, and a prefetching method, based on GPFS hints, targeting to increase small read access performance.

Algorithm 5 Collective Read at Aggregators

```
1: MAPPED_FILE_BLOCK_SET ← map(view_table[mpi_rank], left_view_offset, right_view_offset)
2: for all b in MAPPED_FILE_BLOCK_SET do
3:   if b not in CollBufferPool then
4:     if full(CollBufferPool) then
5:       repl_b ← replacementBlock(CollBufferPool)
6:       if dirty(repl_b) then
7:         write(repl_b)
8:       end if
9:       remove(repl_b,CollBufferPool)
10:    end if
11:    insert(b, CollBufferPool)
12:    if b in PrefetchBufferPool then
13:      copyBlock(b, PrefetchBufferPool, CollBufferPool)
14:      remove(b, PrefetchBufferPool)
15:    else
16:      read(b)
17:    end if
18:  end if
19:  gather(netbuf, b, left_view_offset, right_view_offset, view)
20: end for
21: send(netbuf, mpi_rank)
22: for all b in PrefetchBufferPool do
23:   if gpfs_prefetch_finished(b) then
24:     if full(CollBufferPool) then
25:       repl_b ← replacementBlock(CollBufferPool)
26:       if dirty(repl_b) then
27:         write(repl_b)
28:       end if
29:       remove(repl_b, CollBufferPool)
30:     end if
31:     insert(b, CollBufferPool)
32:     copyBlock(b, PrefetchBufferPool, CollBufferPool)
33:     remove(b, PrefetchBufferPool)
34:   end if
35: end for
36: if not full(PrefetchBufferPool) then
37:   PREFETCH_BLOCK_SCHEDULE ← predict_access(view_table, MAPPED_FILE_BLOCK_SET)
38:   activate_gpfs_prefetching(PREFETCH_BLOCK_SCHEDULE, PrefetchBufferPool)
39: end if
```

Chapter 5

Parallel I/O architecture for supercomputers

The past few years have shown a continuous increase in the performance of supercomputers, as demonstrated by the evolution of Top 500. In November 2009 release while the large majority of the systems (83%) in Top 500 are off-the-shelf clusters, the supercomputers still appear to be more scalable, having 60% share in the Top 50.

Two of the most popular supercomputer architectures are IBM's Blue Gene and Cray's XT. Both Cray and Blue Gene systems scale up to hundreds of thousands of processors. Blue Gene supercomputers have a significant share in the Top 500 list and bring additionally the advantage of a highly energy-efficient solution. Cray has the merit of occupying the first and third place of the November 2009 edition of Top 500.

In order to make full benefit of the processing scalability, the parallel applications need also a scalable parallel I/O system [SHea06, ABB+08]. A limited number of papers have proposed solutions for scalable parallel I/O systems for large supercomputers. Nevertheless, supercomputers have a complex architecture consisting of several networks, several tiers (computing, I/O, storage) and, consequently a potential deep cache hierarchy. This architecture provides a rich set of opportunities for optimizations.

We propose a parallel I/O architecture, which addresses this challenge by proposing a novel scalable parallel I/O solution for supercomputer systems. This solution is based on our generic architecture and consists of a scalable multi-tier caching architecture and a I/O architecture design hiding the latency of file accesses to the applications. In this chapter we describe how the generic parallel I/O architecture for supercomputers can be applied to architectures such as Blue Gene/L and Blue Gene/P.

While the solution proposed in this thesis for supercomputers targets both Blue Gene and Cray systems, being based on components available in both architectures, we mostly concentrate the discussion in this chapter in Blue Gene systems, given the availability of access to the machines through joined projects with Argonne National Laboratory.

5.1 Preliminaries

In this section we summarize basic concepts necessary for understanding the design of our solution for Blue Gene systems: operating system architecture, I/O forwarding, ZeptoOS, and the specific network topology of Blue Gene systems.

5.1.1 Network topology

Blue Gene nodes are interconnected by five networks: 3D torus, tree, global barrier, commodity network, and control. The torus is constructed with point-to-point, serial links between routers, resulting in six nearest-neighbour connections. The bandwidth is shared among the cores. The global collective network (also known as tree network) has its own distinct hardware, which is separate from the torus network. Its topology is a tree; this is a one-to-all, high-bandwidth network for global collective operations, such as broadcast and reductions, and for moving data between the compute and I/O nodes. As with the torus, the cores on a node share this network. The commodity network interconnects I/O nodes and file servers. The global barrier network offers an efficient barrier implementation. The service nodes control the whole machine through the control network. Table 5.1 compares and resumes different features between the Blue Gene/L and Blue Gene/P systems.

Table 5.1: *Comparison of the BG/L and BG/P systems.*

	Property	Blue Gene/L	Blue Gene/P
Node Properties	Node Processors	*2*440 PowerPC*	*4*450 PowerPC*
	Processor Frequency	*0.7GHz*	*0.85GHz*
	Coherency	Software managed	SMP
	L1 Cache (private)	*32KB/processor*	*32KB/processor*
	L2 Cache (private)	14 stream prefetching	14 stream prefetching
	L3 Cache size (shared)	*4MB*	*8MB*
	Main Store/node	*512MB/1GB*	*2GB*
	Main Store Bandwidth	*5.6GB/s (16B wide)*	*13.6GB/s (2*16B wide)*
	Peak Performance	*5.6GF/node*	*13.6GF/node*
Torus Network	Bandwidth	*6*2*175MB/s=2.1GB/s*	*6*2*425MB/s=5.1GB/s*
	Hardware Latency (best)	*200ns (32B packet)*	*160ns (32B packet)*
	Hardware Latency (worst)	*6.4us (64 hops)*	*5us (64 hops)*
Tree Network	Bandwidth	*2*350MB/s=700MB/s*	*2*0.85GB/s=1.7GB/s*
	Hardware Latency	*5.0us*	*4.0us*
Commodity Network	Bandwidth	*1GB/s*	*10GB/s*
System Properties	Peak Performance	*410TF*	*1PF*
	Total Power	*1.7MW*	*2.7MW*

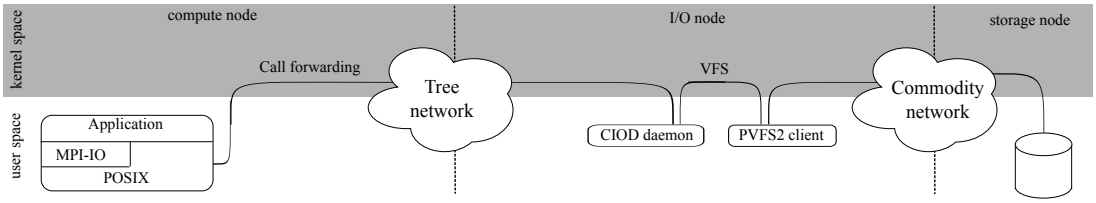

Figure 5.1: *File I/O forwarding for IBM solution.*

5.1.2 Operating system architecture

In the IBM solution [Mea06], the compute node kernel (CNK) is a light-weight operating system offering basic services: creation of one or two address spaces (depending if the running mode is coprocessor or virtual), simple system calls such as setting an alarm, and forwarding I/O-related system calls to the I/O nodes. The I/O nodes run a simplified Linux OS kernel (IOK) with a small memory footprint, an in-memory root file system, TCP/IP and file system support, no swapping, and lacking the majority of classical daemons. The I/O nodes do not run applications. Because the L1 caches are not coherent, all the threads of a daemon run on the same core.

There are several ways to provide I/O support on supercomputers. Applications can use a user-level file system based on FUSE or the SYSIO library [YVC07b] to perform file I/O. The SYSIO library provides POSIX-like file I/O support for remote file systems in user space. Another approach, which is used in the Blue Gene systems, is to forward all I/O requests from the compute nodes to dedicated I/O nodes. The I/O forwarding from compute to I/O nodes is similar to Remote Procedure Calls. The I/O nodes run a fully functional OS kernel and perform I/O on behalf of the compute nodes. This technique, known as *I/O forwarding*, enables applications running on the compute nodes to perform I/O without introducing I/O-specific jitter in the CNK.

As shown in Figure 5.1, the I/O system calls (e.g. file system calls, sockets, etc.) are forwarded through the tree collective network to the I/O node. First, applications access the file system through MPI-IO or POSIX interface. MPI-IO is implemented on top of POSIX file system calls. POSIX calls are forwarded in a RPC-like style to the I/O nodes. Second, the forwarded calls are served on the I/O node by a user-level daemon called CIOD. The CIOD executes the file system call on behalf of the compute node through the VFS interface which communicated with a local PVFS2 client. Finally, the PVFS2 client sends the request to the PVFS2 servers running on the storage nodes. The call return value and data is sent back on the same path in the reverse direction.

The I/O nodes run a simplified Linux OS kernel (IOK) with a small memory footprint, an in-memory root file system, TCP/IP and file system support, and no swapping and lacking the majority of classical daemons. The forwarded calls are served on the I/O node by the control and I/O daemon (CIOD). CIOD executes the requested system calls on locally mounted file systems and returns the results to the compute nodes.

An open-source alternative to the IBM's solution is developed in the ZeptoOS project [hu08]. Under ZeptoOS, Blue Gene compute nodes may run Linux, while the I/O forwarding is implemented in a component called ZOID. The I/O forwarding process from ZeptoOS is similar to the one based on CIOD, in the sense that I/O related calls are forwarded to the I/O nodes, where a multithreaded daemon serves them. However, there are two notable differences in design

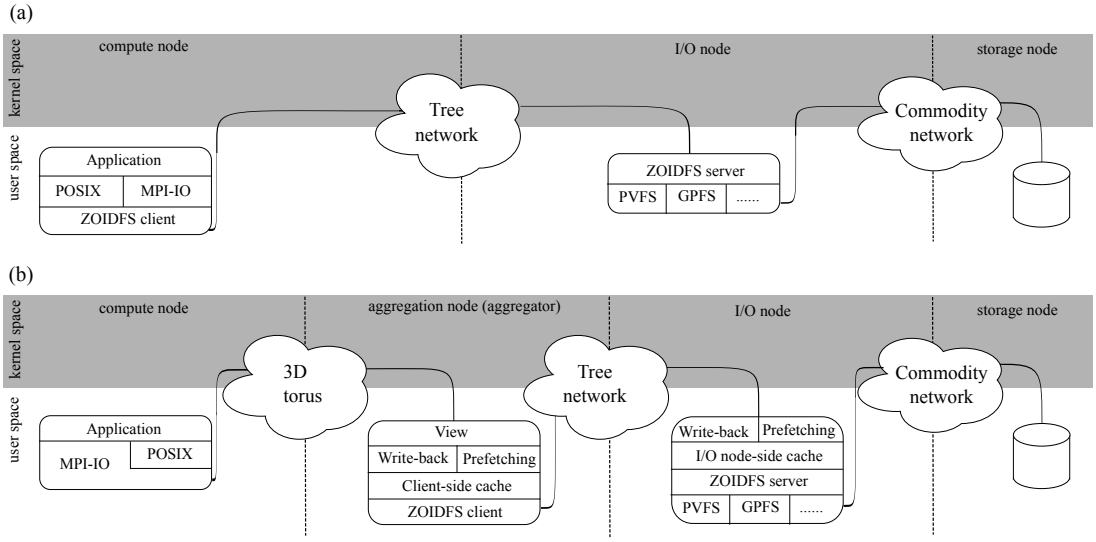

Figure 5.2: *File I/O forwarding in ZeptoOS.*

and implementation between CIOD-based and ZOID-based solutions. First, ZOID comes with its own network protocol, which can be conveniently extended with the help of a plug-in tool, which automatically generates the communication code for new forwarded calls. Second, the file system calls are forwarded through ZOIDFS [IRYB08], an abstract interface for forwarding file system calls. ZOIDFS abstracts away the details of a file system API under a stateless interface consisting of generic functions for file create, open, write, read, close, and so forth.

In Figure 5.2 we represent how file I/O forwarding works in ZeptoOS. For ZOIDFS based solution without caching, MPI-IO and POSIX calls are mapped to abstract file system interface ZOIDFS and forwarded to the I/O nodes. The ZOID daemon acts as a ZOIDFS server and maps ZOIDFS calls onto specific file systems. For a ZOIDFS with client-side and I/O node-side caching solution, applications access the file system through MPI-IO or POSIX interfaces. POSIX may be implemented on top of MPI-IO. The MPI-IO calls are implemented based on MPI communication and are performed in cooperation by aggregators. A client-side cache, write-back and prefetching modules are managed by each aggregator. The aggregation nodes forward ZOIDFS calls through ZOID to the I/O node. I/O node serves the ZOIDFS calls either from the cache or by calling the appropriate file system.

5.2 Architecture description

Our proposed solution is based on a multi-tier architecture depicted in Figure 5.3. The six tiers of the architecture correspond with the generic ones from Figure 3.1: application tier, application I/O forwarding tier, client-side file cache management tier, aggregator I/O forwarding tier, I/O node-side file cache management tier, and storage system tier.

The applications run on a set compute nodes of compute nodes and can be parallel MPI programs or a set of sequential applications. The access to the file system can be done either through MPI-IO or POSIX interfaces. Application I/O forwarding tier has the goal of forwarding the file accesses to the I/O-node side cache module through the scalable torus network. The file

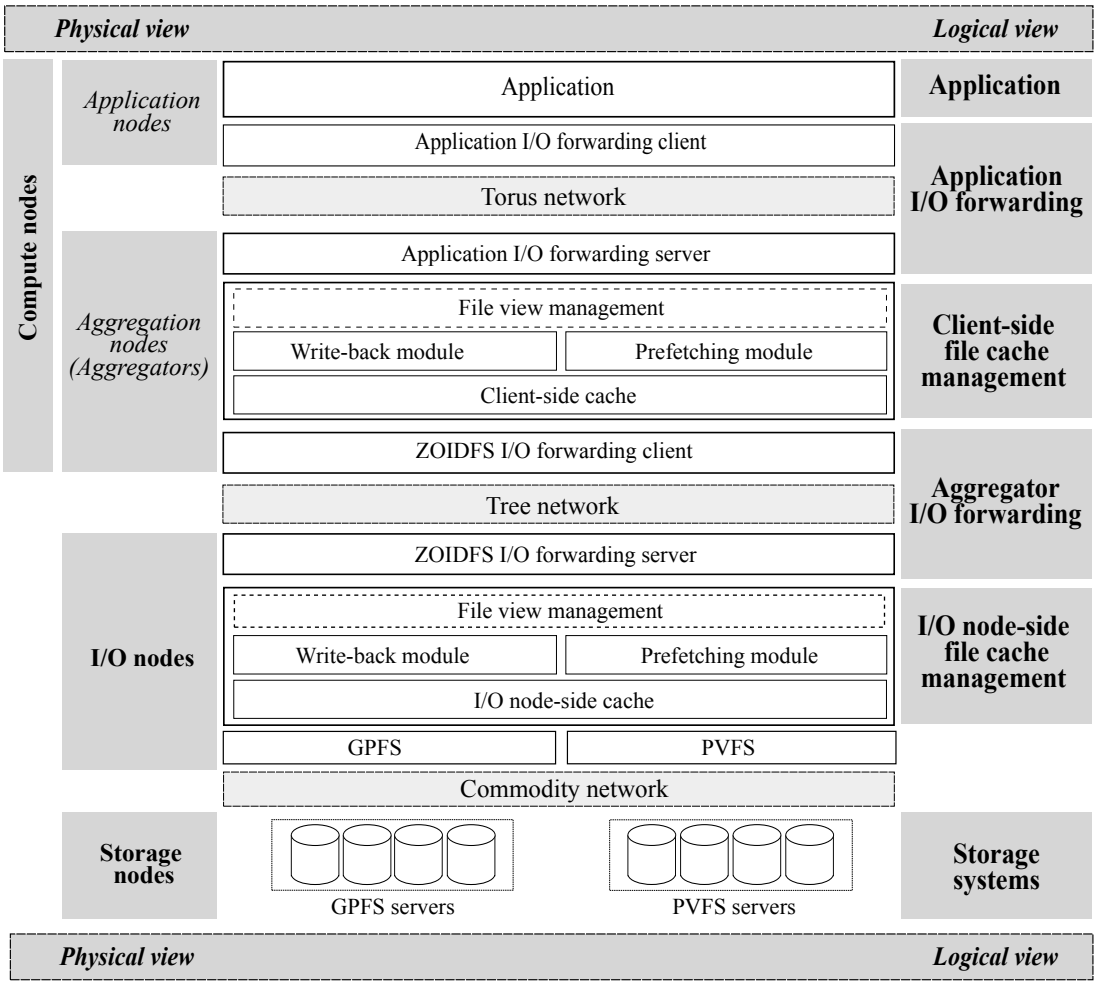

Figure 5.3: *Blue Gene cache architecture organized on six tiers: application, application I/O forwarding, client-side cache, aggregator I/O forwarding, I/O node-side cache and storage. Dotted boxes represent optional modules, which are client and I/O node-side file view management modules.*

access can be done by MPI applications through MPI-IO and by sequential application through POSIX. The POSIX interface is based on FUSE [htt09b], on which its turn resides on MPI-IO.

Client-side file cache management tier helps hide the latency of transfer over the tree network. This approach address the fact that the tree network is shared among all the processes inside a pset and, consequently, may represent a bottleneck if used in an uncoordinated manner. Additionally, when applications running on the allocated partition show temporal locality, the transfers over the tree network may be substantially reduced, in the same manner as in [WFI+09]. Files are mapped round robin over all aggregators in the Blue Gene partition of the application. Each file block is mapped exactly to one aggregator. On its turn the aggregator is mapped to the I/O node corresponding to the pset. Figure 5.4 shows an example of a file mapped on a partition consisting of two psets, with two aggregators per pset. File block 2 can be cached only once at aggregator 2 and at the I/O node 1 and on the storage node 0. The file blocks from aggregators are mapped on the I/O nodes in charge of the corresponding pset. Each I/O node is in charge of caching and accessing the file system blocks of all aggregator of their pset. Note that there may

be exactly one copy of a file block at each level. For instance the file block 2 may be cached only in pset 1 at aggregator 2, I/O node 1 and storage node 0. This figure represents also the transfer pipeline of accessing the file.

The application accesses the client-side file cache through the torus network. The client-side file management layer is file system independent: ZOIDFS is a virtual file system layer, that also provides I/O forwarding (as described in Section 5.1.2). The ZOIDFS I/O forwarding client resides in the aggregator I/O forwarding tier. This tier has the goal of forwarding the file accesses to the next tier through the tree network. The forwarding is done on-demand, when the aggregator issues the file access. The file access is done by ZOIDFS client (explained in details in Section 5.1.2).

I/O node-side file cache management tier is designed by decoupling the communication between compute node and I/O node over the tree network from the transfers to the file systems. The I/O node-side cache absorbs the blocks transfered asynchronously from the client-side cache, and hides the transfers between the I/O nodes and file systems. The I/O node-side file cache management layer is managed by the ZOID daemon running on each I/O node. The daemon receives ZOIDFS requests from the compute nodes and serves them from the cache. The communication with the compute nodes is decoupled from the file system access, allowing for a full overlap of the two operations. As in the case of the generic client-side file cache management tier, data staging is managed in two modules operating on a file cache: write-back and prefetching modules. Each of these modules acts in coordination with the corresponding module from the compute nodes, and therefore, are units in the file write and file read pipelines.

The storage system consists of file system servers running on storage nodes and accessing disks over a storage area network. For the solution presented here, the access to the storage system is transparent, it is the file system mounted on the I/O nodes that is in charge of communicating with the storage system.

5.3 Data staging

In Blue Gene systems file system access implies pipelining data through three different networks (tree, commodity and storage network) from compute nodes through I/O nodes and storage servers to finally reach the storage. The I/O forwarding process from our architecture is shown in Figure 5.2).

A bottleneck in any of these networks may be propagated up to the applications. The data staging is designed to hide the latency of the transfers through these networks, and, therefore, reduce the probability of applications perceiving I/O congestions. Our solutions addresses two potential hot spots in the Blue Gene architecture: the tree network and the file system. The tree network is especially problematic, given its shared use by all the processors in a pset. Additionally, the file systems are shared by the whole system and may unexpectedly slow down the access of a particular partition when data intensive applications are run in other partitions.

In the following subsections we describe the specific I/O node write-back and prefetching modules for supercomputers.

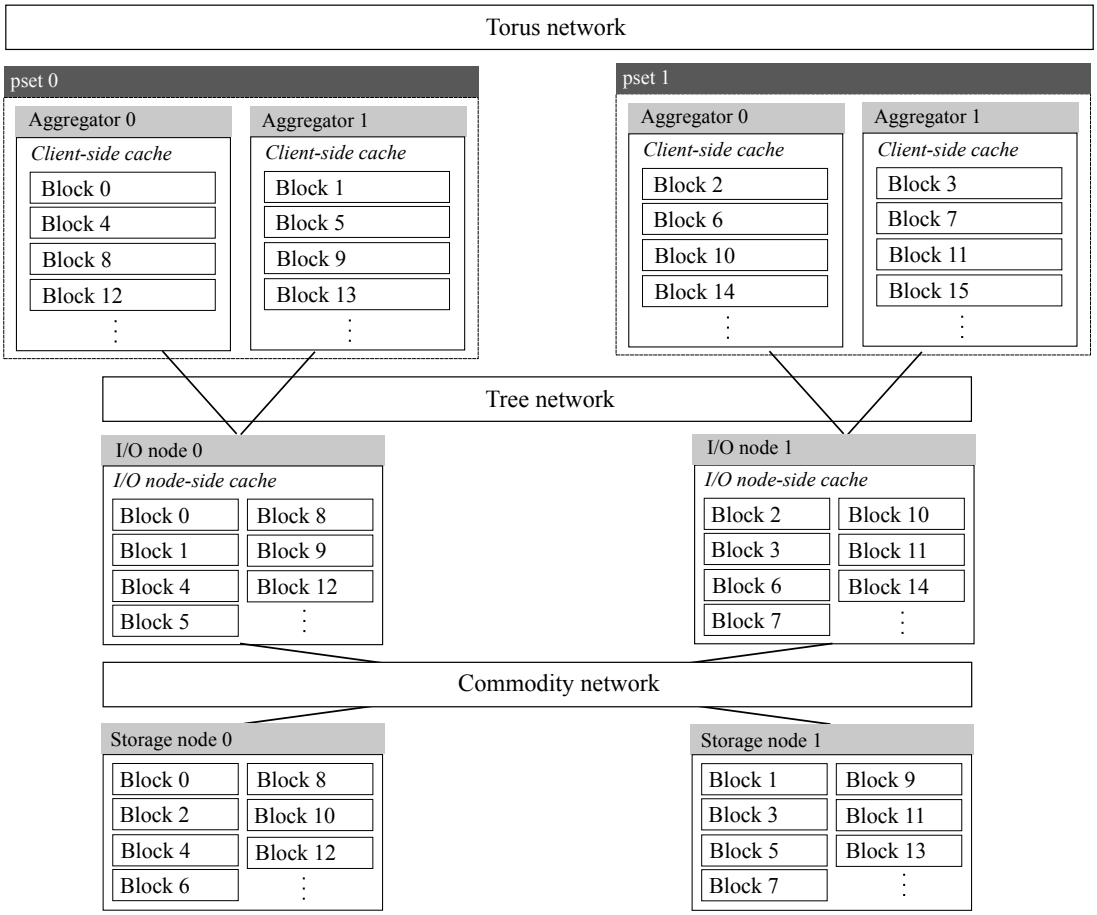

Figure 5.4: *File mapping example in Blue Gene.*

5.3.1 I/O node-side write-back

ZOIDFS file write-back operation consists of the following steps. At access time, a compute node sends to all aggregators only the offset and count of the file access: offset (*offset*) and count (*count*). The file cache is represented by the *FileHash* variable. Subsequently, the ZOIDFS I/O thread runs the Algorithm 6 for each aggregator request received in the input buffer. First, the set of blocks to be written (*MAPPED_FILE_BLOCK_SET*) is calculated by mapping the access offset onto file blocks. For each block b from *MAPPED_FILE_BLOCK_SET* (line 3), if the block is not in the file hash (line 4) and if the file hash is full (line 5), a file block *repl_b* is chosen for replacement by a LRU policy (line 6), flushed to the storage system (line 8) if dirty, and removed from the hash (line 10). Finally, b is inserted into the file hash (line 12). At this point, data are scattered from ZOIDFS buffer into b (line 14) and b is marked as dirty (line 15).

The I/O node write-back policy is enforced in an I/O thread shown as Algorithm 7. The I/O thread is activated, when a high water mark of dirty blocks is reached. While the number of dirty blocks is higher than a low water mark, the thread flushes buffers to the storage, choosing them by employing Last Recently Modified (LRM) policy. Finally, the I/O thread goes to sleep until the high water mark is reached again.

The write-back allows for overlapping the computation and I/O, by gradually transferring

Algorithm 6 I/O node write-back: ZOIDFS thread-side

```
1: MAPPED_FILE_BLOCK_SET ← map(count, offset, block_size)
2: for all b in MAPPED_FILE_BLOCK_SET do
3:   if b not in FileHash then
4:     if full(FileHash) then
5:       repl_b ← replacementBlock(FileHash)
6:       if dirty(repl_b) then
7:         write(repl_b)
8:       end if
9:       remove(repl_b, FileHash)
10:    end if
11:    insert(b,FileHash)
12:  end if
13:  scatter(netbuf, b, left_view_offset, right_view_offset, view)
14:  set_dirty(b)
15: end for
```

Algorithm 7 I/O node write-back: I/O thread-side

```
1: while 1 do
2:   if no_of_dirty_buffers>high_water_mark then
3:     sleep
4:   end if
5:   while no_of_dirty_buffers > low_water_mark do
6:     b ← chooseFlushBlock(FileHash)
7:     write(b)
8:     set_nodirty(b)
9:   end while
10: end while
```

the data from the I/O nodes to the storage servers. This approach distributes the cost of the file access over the computation phase and communication between compute nodes and I/O nodes. Therefore, a two-level asynchronous file write strategy is enforced: the compute nodes asynchronously write back data to the I/O nodes, while the I/O nodes write asynchronously back data to the storage system.

5.3.2 I/O node-side prefetching

The objective of I/O node prefetching is to hide the latency of read operations by predicting the future accesses of compute nodes.

The prefetching mechanism is leveraged by each I/O thread running at the I/O node. The prediction is based on the round-robin buffer distribution over the aggregators. Given that all collective buffers of an aggregator are mapped on the same I/O node, we propose a simple strided prefetching policy. When an on-demand or a prefetch read for a collective block corresponding to an aggregator returns, a new prefetch is issued for the next file block of the same aggregator. The prefetching is performed by the I/O thread, while on-demand reads are issued by the ZOIDFS

thread serving the requesting aggregator. A cache block, for which a read request has been issued, is blocked in memory, and all threads requesting data from the same page block at a condition variable. A further relaxation of this approach by allowing reads from page hits to bypass reads from page misses is possible, but it has not been implemented in the current prototype.

At access time, an aggregator sends to the I/O node only the offset and count of the file access: access offset (*offset*) and count (*count*).

Algorithm 8 I/O node prefetching: ZOIDFS thread-side

```
1:  MAPPED_FILE_BLOCK_SET ← map(count, offset, block_size)
2:  for all b in MAPPED_FILE_BLOCK_SET do
3:    if b not in FileHash then
4:      if full(FileHash) then
5:        repl_b ← replacementBlock(FileHash)
6:        if dirty(repl_b) then
7:          write(repl_b)
8:        end if
9:        remove(repl_b, FileHash)
10:     end if
11:     insert(b, FileHash)
12:     if b in PrefetchHash then
13:       remove(b, PrefetchHash)
14:     else
15:       read(b)
16:     end if
17:   end if
18:   gather(netbuf, b, offset, count)
19: end for
20: if not full(PrefetchHash) then
21:   PREFETCH_BLOCK_SCHEDULE ← predict_access(crt_blk_addr +
      no_of_aggregators * block_size)
22:   insert(PREFETCH_BLOCK_SCHEDULE, PrefetchHash)
23:   signal_io_thread()
24: end if
```

Subsequently, each ZOIDFS thread runs Algorithm 8 for each aggregator request. First, the set of blocks to be read from, (*MAPPED_FILE_BLOCK_SET*) is calculated by mapping the access interval onto file blocks through the offset and count of file access and the block size, *block_size* (line 1). For each block *b* from *MAPPED_FILE_BLOCK_SET* (line 2), if the block is not in the file cache (line 3), it is inserted (line 12), after replacing a block if the hash was full (lines 6-12). If *b* has been already read into prefetching hash (line 13), i.e. has been scheduled for prefetch previously, the future access is removed from prefetching hash (line 21). If *b* has not been scheduled for prefetching, it is read directly from the storage node (line 13). At this moment, the requested data can be gathered into the response buffer (*buffer*) through the offset from file cache buffer *b* (line 16). Finally, if the prefetching hash is not full, it is filled by predicting future block accesses based on the round-robin collective buffer distribution over the aggregators (line 20) and scheduling them for prefetching (line 21).

The I/O thread at I/O nodes runs Algorithm 9. All blocks that have been scheduled for prefetching (line 1) are moved to the file cache (lines 11-12). If the file cache is full, block replacement is performed (lines 2-10).

Algorithm 9 I/O node prefetching: I/O thread-side

```
1:  for all b in PrefetchHash do
2:    if full(FileHash) then
3:      repl_b ← replacementBlock(FileHash)
4:      if dirty(repl_b) then
5:        write(repl_b)
6:        set_nodirty(repl_b)
7:      end if
8:      remove(repl_b, FileHash)
9:    end if
10:   read(b)
11:   insert(b,FileHash)
12:   copyBlock(b, PrefetchHash, FileHash)
13:   remove(b, PrefetchHash)
14: end for
```

5.4 Discussion

Blue Gene/L presents two limitations that affect the efficiency of our solution. First, the compute nodes do not support multi-threading. Therefore, the data flushing from the collective buffers cannot be flushed asynchronously to the I/O nodes. Second, because the L1 caches are not coherent, the I/O nodes run all the ZOID threads on the same processor. Therefore, it is not possible to overlap communication and file transfer. However, our solution allows overlapping computation and file transfer: while the applications are running, the file cache on the I/O node is asynchronously flushed to the file system over the commodity network.

The limitations for Blue Gene/L have been removed from the Blue Gene/P architecture: Blue Gene/P has limited multi-threading support, but adequate for our caching strategies and the L1 caches of the cores are coherent. Therefore, we expect that our solution will offer an additional performance benefit on Blue Gene/P systems.

5.5 Summary

In this chapter we have described the application of a multiple-level cache architecture to Blue Gene systems. The solution includes a compute- and I/O-node write-back policy, which asynchronously transfers data from the compute nodes to the I/O nodes. Additionally, the solution includes a two-level prefetching strategy, which asynchronously transfers file data from the file system to the I/O node based on mapping of aggregators to I/O nodes and from I/O nodes to the compute nodes based on application views. Both the data-staging and prefetching strategies provide a significant performance benefit, whose main source comes from the efficient utilization of the Blue Gene parallelism and asynchronous transfers across file cache hierarchy.

The client-side and I/O node-side modules are generic and portable, the implementation can be used unmodified on clusters or any other supercomputers. This can be achieved either by extending the ZOID back-end to these systems or, alternatively, through an ADIO module for file systems mounted on the I/O nodes.

Chapter 6

Evaluation

The previous chapters showed the generic parallel I/O architecture and all techniques and policies introduced in order to provide flexible and cost-effective levels of performance and scalability. The goal of this chapter is an experimental evaluation of the performance and scalability of our parallel I/O architecture and a quantitative comparison with another existing available methods. This chapter focuses on the evaluation of the implemented prototypes based on our solutions for clusters and supercomputers.

The chapter is organized in four sections. First we describe the benchmarking toolkit used to evaluate the performance of our solutions. Second, we analyze the performance of our proposed architecture on clusters, including different solutions presented in Chapter 4. Third, we evaluate our architecture on supercomputers (presented in Chapter 5) in terms of scalability and performance. Finally, we discuss the conclusion of our experiments.

6.1 Benchmarking setup

In this section we present the collection of benchmarks used, in order to analyze the performance of our generic architecture and to demonstrate its advantages on various aspects such as performance and scalability.

NASA's BTIO. NASA's BTIO benchmark [WdW03] solves the Block-Tridiagonal (BT) problem, which employs a complex domain decomposition across a square number of compute nodes, as shown in the Figure 6.1 for 9 processes. Each compute node is responsible for multiple Cartesian subsets of the entire data set. The execution alternates computation and I/O phases. Initially, all compute nodes collectively open a file and declare views on the relevant file regions. After each five computing steps the compute nodes write the solution to a file through a collective operation. At the end, the resulted file is collectively read and the solution verified for correctness. The benchmark reports the total time including the time spent to write the solution to the file. However, the verification phase time containing the reading of data from files is not included in

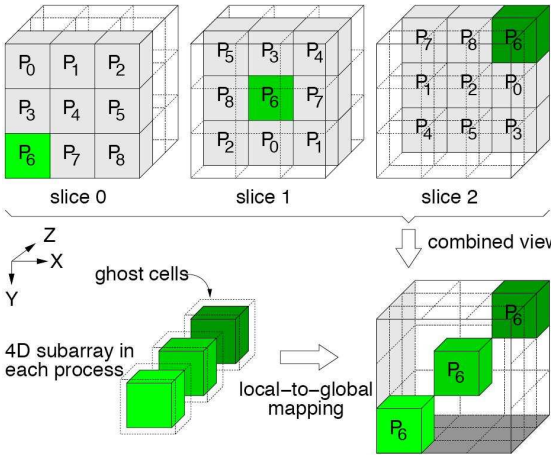

Figure 6.1: *BTIO data decomposition for 9 processes.*

Table 6.1: *Granularity of BTIO file accesses.*

Process count	Class B	Class C
9	1360	2160
16	1000	1640
25	800	1280
36	680	1080
49	600	920
64	480	800

the reported total time. The access pattern of BTIO is nested-strided with a nesting depth of 2 with the file access granularity given in Table 6.1.

MPI Tile I/O. MPI Tile I/O [MPI] is an MPI-IO benchmark testing the performance of non-contiguous data access. In this application, data I/O access is non-contiguous and is issued in a single step by using collective I/O. It tests the performance of concurrently accessing a two-dimensional dense data set, simulating the type of workload that exists in some visualization and numerical applications.

ROMIO test suite. The ROMIO test suite consists of a number of correctness and performance tests. The collperf.c test measures the I/O throughputs for both file read and write operations for accessing a three-dimensional block-distributed array. The partitioning of data is done through the assignment of a number of processes on each Cartesian dimension. The arrays are first written to a file, then read back into the compute nodes and finally the throughput is reported.

gpfsPerf. The gpfsPerf is an IBM benchmark from the GPFS library distribution. GpfsPerf writes and reads a collection of fixed-size records to/from a file with three different types of access patterns: sequential, strided, and random. The user can choose to enable or disable the data-shipping optimization through a specific `fcntl` operation provided by GPFS library [PTH+00].

FLASH I/O. FLASH I/O [FOR+00] benchmark simulates the I/O pattern of FLASH. The FLASH code is an adaptive mesh refinement application that solves fully compressible, reactive hydrodynamic equations. The benchmark recreates the primary data structures in the FLASH code and produces a checkpoint file, a plot-file for centered data, and a plot-file for corner data, using parallel netCDF. The access pattern is non-contiguous both in memory and in file, making it a challenging application for parallel I/O systems.

SimParIO. We have implemented a benchmark called SimParIO, that simulates the behaviour of scientific parallel applications. The main goal of SimParIO was to facilitate the evaluation of the degree of overlapping between computation and parallel I/O. SimParIO consist of alternating compute phases and I/O phases. In SimParIO, the compute phases are simulated by idle spinning. The length of a compute phase is user-configurable. Each compute phase is followed by an I/O phase, in which all processes write to, or read from, non-overlapping regions of the file. The access pattern can be contiguous or single strided.

6.2 Evaluation on clusters

In this section we perform experiments in order to evaluate the performance of our multi-tier cache I/O architecture on clusters. This section is structured as follows. First, we show the performance results for view-based I/O synchronous solution (presented in Chapter 3). Second we evaluate AHPIOS comparing different measures such as scalability and performance. Finally, we compare our asynchronous view-based I/O and GPFS-based I/O solutions.

6.2.1 View-based I/O

We evaluate view-based I/O as an one-level optimization on an existing cluster infrastructure.

The evaluation of our implementation for view-based I/O was performed on NEC Cacau Xeon EM64T cluster at HLRS Stuttgart. This cluster has the following characteristics: 200 Intel Xeon EM64T CPU's (3.2GHz), 160 nodes with 1GByte of RAM memory and 40 nodes with 2GBytes of RAM memory, and an Infiniband network, which interconnects the compute nodes.

The parallel file system has been PVFS version 2.6.3, running 8 servers, from which all 8 were I/O nodes and one metadata node. The files of PVFS were striped over all I/O servers round-robin with a block of 64KBytes. The PVFS servers and the application compute nodes were running on disjoint nodes. The communication protocol of PVFS was TCP/IP on top of the native Infiniband communication library. The MPI distribution was MPICH2 1.0.5. We ran all tests with one process per compute node.

View-based I/O as well as two-phase I/O had a network buffer of 4 MBytes. In addition, view-based I/O had a collective buffer pool of maximum 64MBytes. We present the results for three benchmarks: NASA'S BTIO, MPI-Tile-IO and collperf.

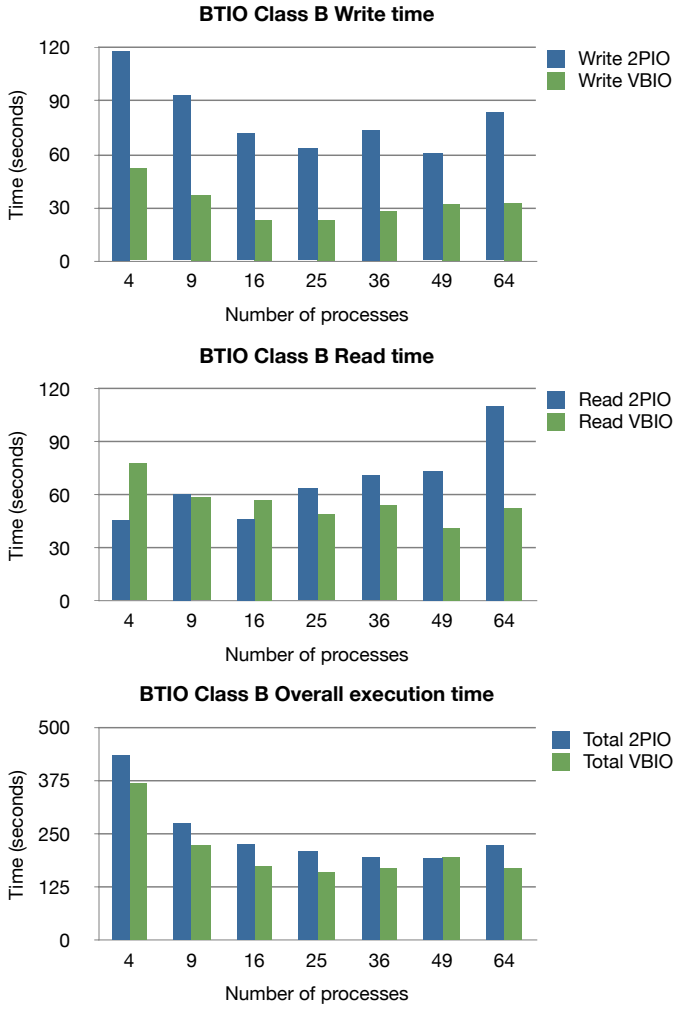

Figure 6.2: *View-based I/O. BTIO performance for class B. The first plot shows the total time (in seconds) spent in writing the file. The total time spent in reading from the file is plotted in the second row. Finally, the last graph depicts the final time as reported by the application.*

BTIO benchmark

This subsection evaluates and compares the performance of the ROMIO-based collective operations.

BTIO explicitly sets the size of the collective buffer to 1MByte and assigns all compute nodes for two-phase I/O aggregators or view-based I/O aggregators.

It is important to note here that at this point, we evaluate first a synchronous version in which the write operations of view-based I/O are flushed to the file system when the distributed cache on compute nodes is full or the file is closed. In Section 6.2.3 we evaluate a full version of view-based I/O, including the data staging policies.

Figures 6.2 and 6.3 compare the time results of view-based I/O and two-phase I/O. The plots show the results for classes B and C, respectively. The first plot shows the total time (in seconds) spent in writing the file. The total time spent in reading from the file is plotted in the second row. Finally, the last graphs depicts the final time as reported by the application,

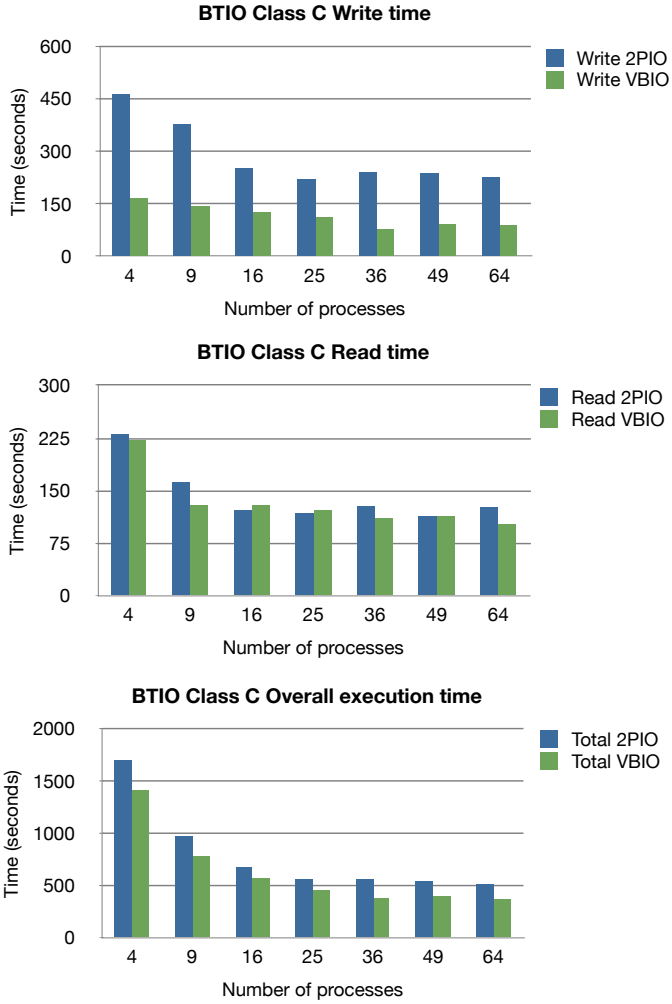

Figure 6.3: *View-based I/O. BTIO performance for class C. The first plot shows the total time (in seconds) spent in writing the file. The total time spent in reading from the file is plotted in the second row. Finally, the last graph depicts the final time as reported by the application.*

including the computation and the times to open the file, set the view, write the file and close it (the read time is not contained).

We observe that view-based I/O outperforms the native approach in most of the cases. Additionally, the improvement increases with the number of compute nodes.

This performance improvement can be attributed to several causes: the reduction of the transferred offset-length lists, the improvement of scatter-gather operations, the smaller number of collective interdependencies, the usage of a distributed cache. In view-based I/O, the collective buffers are cached at the aggregators across collective I/O operations. If the caches are large enough, subsequent read operations will find the buffers in the cache.

We can observe that view-based I/O was more effective for both problem sizes. In BTIO class B, writes were between 89% and 121% faster and reads were between 3% to 109% faster than two-phase. Compared with BTIO class C, the results show that our approach was between 25% to 310% faster for writes and between 4% to 23% for reads. As explained earlier, BTIO application

reports a total time including the write and file close time (the read time is not included). However, if we add the read time to the overall execution time, view-based reduced its execution time by 8% to 50% for class B and 14% to 38% for class C. Furthermore, a significant time of BTIO execution is spent in compute phases (around 80%), therefore, the actual improvement of the I/O phase is even more significant.

Table 6.2 lists the overhead of length-offset transfers. The table shows the total amount of transferred lists (in bytes) by two-phase I/O and view-based I/O. BTIO performs 40 write and read iterations, thus, in two-phase I/O the lists of length-offset pairs are transferred 80 times. On the other hand, in view-based I/O, the lists of length/offset pairs are generated and transferred twice. The table shows that, for 64 processes, two-phase I/O transfers ten times more than view-based I/O.

Table 6.2: *The overhead of lists of length-offset pairs (in KBytes) of BTIO Class B.*

N. Processes	Two-phase I/O	View-based I/O
25	8128	229
36	9753	475
49	11379	881
64	13005	1504

Figure 6.4 gives more insight about the execution of two-phase I/O and view-based I/O, by showing the breakdowns of the total time spent in computation, communication, and file access of collective write and read operations, for class B from 4 to 64 processes.

For write, two-phase I/O is largely affected by communication in write and read operations (up to 86% in overall execution time). On the other hand, view-based I/O spent less time for communication, reporting maximum times of 31.4 seconds, as opposed to two-phase I/O, that spent 62.4 seconds. For two-phase I/O, the parallel write time is 84.5 seconds, of which 25.3% are spent in computation and 72.4% in communication. View-based I/O needed only 31.5 seconds, of which 1.0% are spent in computation and 99.0% in communication.

For read, view-based I/O employed the most time in communication. View-based I/O spent 26.5 seconds in communication while two-phase I/O spent 53.4 seconds. Our implementation brings down the cost of communication dramatically. As well, view-based I/O eliminates the contention of metadata operations.

When writing, notice that, starting with 36 processes, there is no file access time in the breakdown. As we explained, this is due to the fact that all the data fit in the cache and is flushed to the file system at close time.

MPI Tile I/O benchmark

This subsection evaluates and compares the aggregate throughput (in MBytes/second) of write and read collective operations of MPI Tile I/O.

In our experiments, each process renders a 1x1 tile with 2048x1536 pixels and the size of each element is 8 bytes. Figure 6.5 shows the read and write throughputs with the MPI Tile

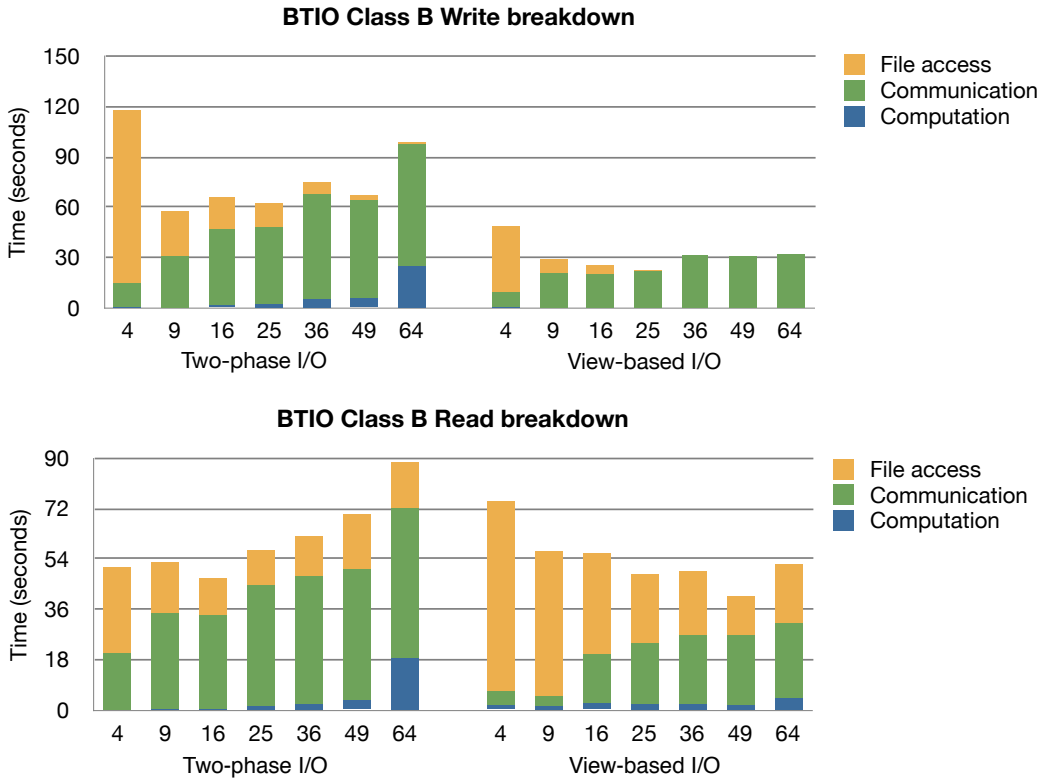

Figure 6.4: *View-based I/O. Breakdowns of the total time spent in computation, communication and file access of collective write and read operations, for two-phase I/O (2PIO) and view-based I/O (VBIO), for class B from 4 to 64 processes. VBIO brings down the cost of communication and eliminates the contention of metadata operations.*

I/O benchmark on Cacau. In this case, the size of the file was kept constant and the number of processes was varied, leading to a file size of 1.5GBytes per execution.

Figure 6.5 shows the results of write and read throughput, respectively. Also, the bottom graph depicts the time spent (in seconds) in file closing compared with writing time of the file. It is important to notice here that, the collective buffers are not cached at the aggregators across collective I/O operations, because this benchmark closes the file between write and read operations. Subsequent read operations do not find data in the cache; thus, for this benchmark, view-based I/O does not benefit from cached collective buffers. In spite of that, view-based I/O performs considerable better than two-phase I/O, for both reads and writes. The upper graph shows that, for a large number of processes, the benchmark reaches file write throughputs of up to 570 MBytes/sec. However, it is important to note that for 36 processes, the aggregators stored all the data into the distributed cache, and, subsequently, did not flush data to the file system until the file is closed. The lower graph shows the overall time spent in write and close operations and we note that the view I/O performs better in three of the four scenarios. As shown in the graph, when compared to two-phase I/O, view-based I/O improves MPI Tile I/O write time between 44% and 59%.

Although read operations do not take advantage of distributed cache, the view-based I/O performance is better than the one for two-phase I/O in all cases. The MPI Tile I/O benchmark read performance results show that our approach attains a 87% improvement compared to two-

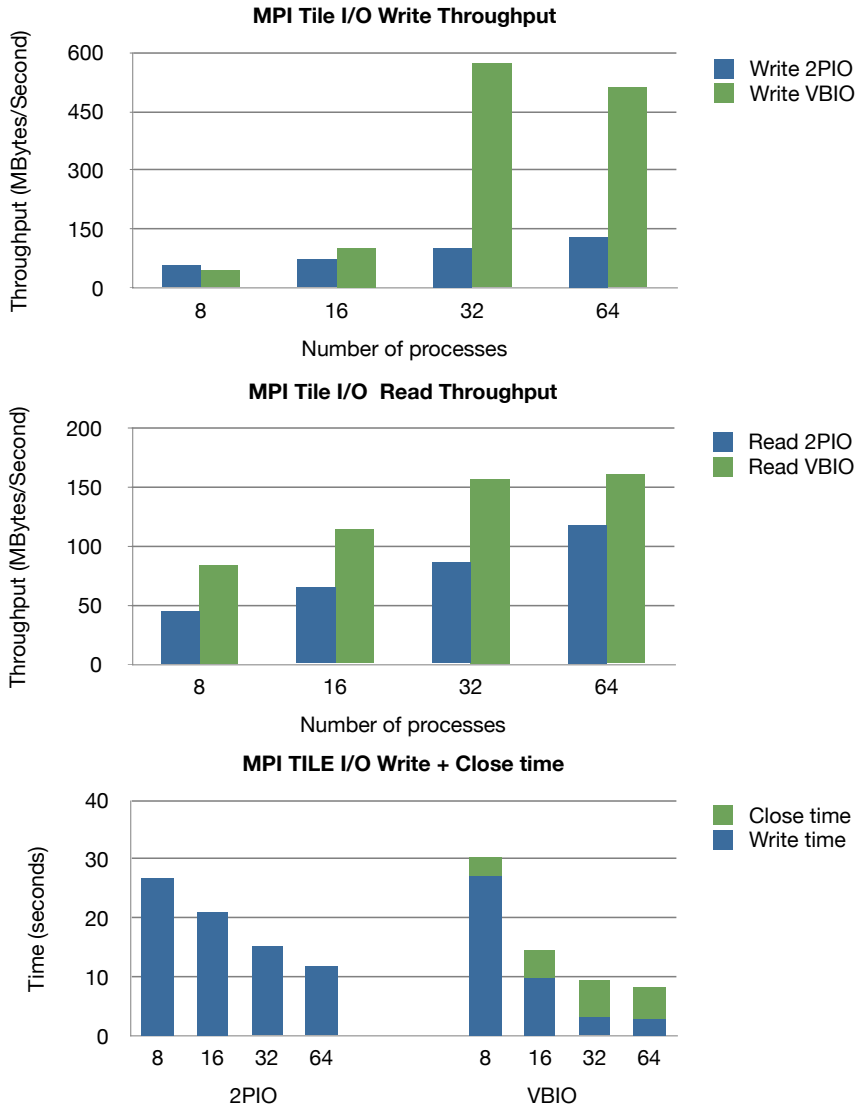

Figure 6.5: *View-based I/O. MPI-TILE-IO results of write and read throughput, for two-phase I/O (2PIO) and view-based I/O (VBIO). Although read operations do not take advantage of the distributed cache, the VBIO performance is better than the one for 2PIO in all cases.*

phase I/O for eight compute nodes and a 36% for 64 compute nodes. This is due to the efficient implementation of views and non-contiguous accesses.

Collective performance benchmark

In this subsection we report the aggregate file read and write throughput of a collective I/O benchmark from the ROMIO test suite.

For ROMIO-based collective I/O, all the compute nodes play the roles of aggregators and the size of the collective buffer is 4MBytes (the default behavior). The network buffer and the stripe size of view-based is set to 4MBytes, as well. The three-dimensional array tested, has 512 x 512 x 512 elements, with an element size of an integer (4 bytes), resulting in a file of 512MBytes.

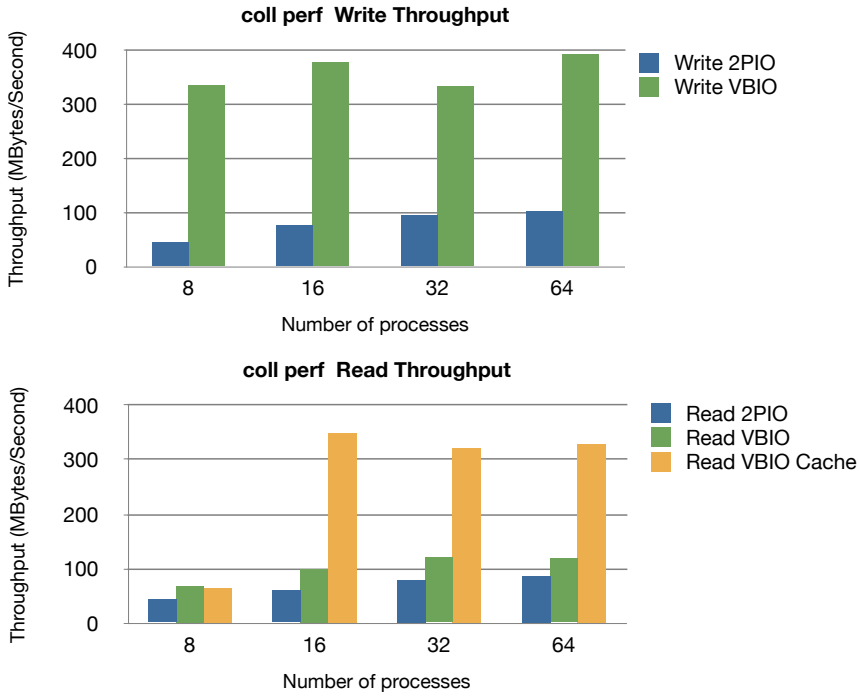

Figure 6.6: *View-based I/O. Coll perf performance. In order to show the effect the distributed file cache, we have also evaluated a slightly modified version of this benchmark, which does not close the file between writes and reads (shown as CB in the graphs).*

The measurements for 8, 16, 32, and 64 processes are plotted in Figure 6.6. By default, the benchmark closes the file between the write and read operations. In order to show the effect of the distributed file cache, we have also evaluated a slightly modified version of this benchmark, which does not close the file between writes and reads (shown as CB in the graphs).

As shown in Figure 6.6, compared to two-phase I/O, view-based I/O improves collperf write throughput by 286% and 633%. On the other hand, collperf read throughput is improved by about 36% and 66% without cache effects and by about 44% and 445% with it.

In order to understand better the results, we traced the collperf benchmark by using the library MPE [GKL95]. We picked up the execution with 8 and 64 processes, because these were the cases for which the performance differed significantly. Table 6.3 shows the number of point-to-point and collective MPI calls of all processes.

We note that the number of collective collective communications and synchronization operations performed by the two-phase implementation is considerably higher. In two-phase I/O the collective buffering is done at compute nodes, which need to perform expensive all to all operations in the shuffle phase: first of all in order to get the list of file offset-length pairs and then in order to get the data.

Discussion

Our experimental results show that synchronous view-based I/O can significantly reduce the total run time of a data intensive parallel application, by reducing both I/O cost and implicit synchronization cost, while requiring no extra communication time. For example, view-based I/O

Table 6.3: *The count of MPI point-to-point and collective communications in collperf benchmark. The table compares the results for two-phase I/O (TPIO) and view-based I/O (VBIO) for 8 and 64 processes.*

MPI call	2PIO 8 CN	VBIO 8 CN	2PIO 64 CN	VBIO 64 CN
MPI_Send	0	0	0	0
MPI_Recv	0	512	0	12288
MPI_Isend	288	576	2560	16384
MPI_Irecv	288	64	2560	4096
MPI_Waitall	656	8	9472	64
MPI_Bcast	528	0	64	0
MPI_Barrier	528	16	8448	128
MPI_Allreduce	61	16	962	128
MPI_Alltoall	528	0	962	0
MPI_Allgather	32	0	8448	0

reduced the BTIO overall execution time by 40%. The MPI-Tile-IO performance results prove that view-based I/O outperforms two-phase I/O, even without caching the collective buffers. Finally, the benchmarks show that, the write-on-close approach brings satisfactory results in all cases.

6.2.2 AHPIOS

Our goal in this evaluation is to demonstrate that, by the tight integration between application and library offered by the full-AHPIOS solution, a significant performance improvement can be obtained using AHPIOS. In the following benchmarks we compare five different solutions for parallel I/O access: ROMIO two-phase I/O over PVFS2 (2PIO-PVFS2), ROMIO two-phase I/O over Lustre (2PIO-Lustre), ROMIO two-phase I/O over AHPIOS (2PIO-IONS) and the two AHPIOS-based solutions: server-directed I/O (VBIO-IONS) and client-directed I/O (VBIO-CS-IONS).

The evaluation presented was performed on the "Lonestar" parallel computer at Texas Advanced Computing Center (TACC) [Lon], which is part of the Teragrid framework [Ter]. A node consists of a Dell PowerEdge 1955 blade running a 2.6 x86_64 Linux kernel. Each node contains two Xeon Intel Duo-Core 64-bit processors on a single board. The Core frequency is 2.66GHz and supports 4 floating-point operations per clock period with a peak performance of 10.6 GFLOPS/core or 42.6GFLOPS/node. There is a 8GBytes memory in each node. The interconnect is an Infiniband with a fat-tree topology. The employed MPI library is MPICH2 version 1.0.5, with the communication running over TCP/IP sockets. The Lonestar Storage includes a 73GBytes SATA drive (60GBytes usable by user) on each node (the I/O servers of AHPIOS and PVFS2 used this storage). The work file system, also accessible from all nodes, is a Lustre

parallel file system with 68TBytes of DataDirect Storage.

We compared AHPIOS with Lustre, the parallel file system installed on Lonestar, and with PVFS2, which we launched through the batch system, before the application is started. AHPIOS and PVFS2 used 8 I/O nodes. For AHPIOS and PVFS2, the I/O servers and application processes are running on disjoint nodes. Lustre is installed over 32 Object Storage Targets and stripes by default files over 8 consecutive OSTs, chosen through an algorithm that combines the randomness with load-balance awareness. Lustre uses the buffer cache of the compute nodes up to a maximum of 6912MBytes per node. Additionally, Lustre has an aggressive prefetching policy, which reads ahead up to 40 MBytes. AHPIOS employs a distributed file cache managed by application processes for client-directed I/O. PVFS2 and AHPIOS with server-directed I/O do not cache data at the clients.

On Lonestar the internal communication of Lustre is performed directly over the Infiniband network. PVFS2 setup could only run TCP/IP over Infiniband. The MPI communication employed by AHPIOS was performed with MPICH2 and not with the Infiniband MVPICH, and consequently, also over TCP/IP. Therefore, in all performed measurements, Lustre has an advantage over PVFS2 and AHPIOS, due to its considerably lower communication costs (TCP/IP sockets are known for high overhead).

Scalability

We evaluated the scalability of independent I/O operations on AHPIOS.

The processes of an MPI program write and read in parallel disjoint contiguous regions of a file stored over an AHPIOS system for different numbers of AHPIOS servers. In this experiment the first level distributed file cache is disabled. We evaluate two scenarios: one in which the compute nodes have also local storage and the AHPIOS servers run on the same nodes as the MPI processes and another one in which MPI processes and AHPIOS servers run on disjoint sets of nodes.

Figure 6.7 shows the aggregate I/O throughput for n MPI processes writing and reading to/from an AHPIOS partition with n AHPIOS servers. The figure represents the throughput to the AHPIOS servers. We can see that the file access performance scales well with the partition size. This happens independently of the location of AHPIOS servers, both when the AHPIOS servers run on the same nodes as the MPI application and on disjoint nodes.

Additionally, in Figure 6.7 the number of MPI processes running on distinct nodes is $n{=}64$ and the number of AHPIOS server is varied between 1 to 64. We notice that the aggregate throughput of both write and read operations scales smoothly with the number of AHPIOS servers, as in the previous case, independently if the AHPIOS servers share or not the compute node with the MPI application.

BTIO benchmark

In this subsection we report the results for the MPI implementation of the benchmark, which uses MPI-IO's collective I/O routines. The collective I/O routines provide significantly better results than the independent ones, due to coalescing of small requests and a more efficient usage of the network and disk transfers. On all runs we set the benchmark to execute 25 compute steps, which correspond to 5 I/O steps (5 collective writes followed by 5 collective reads). The

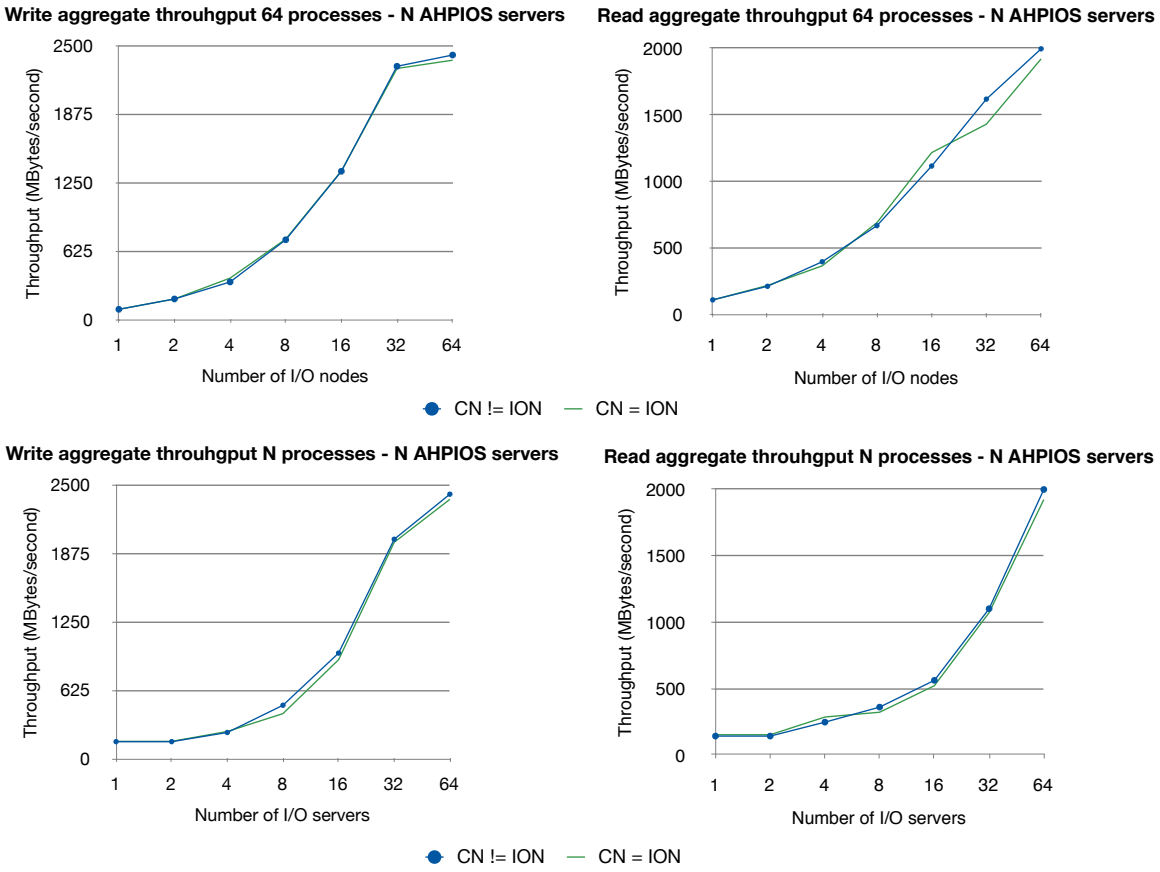

Figure 6.7: *AHPIOS scalability. Figure shows the aggregate I/O throughput for* n *MPI processes writing and reading to/from an AHPIOS partition with* n *AHPIOS servers.*

benchmark does not explicitly commit the file writes to disks.

Figures 6.8 and 6.8 show the results for the BTIO classes B and C. Because the latency is hidden by AHPIOS at file access time, we show the file write time on the first row, the file read time on the second , and finally the total time as reported by the benchmark (includes file open, set view, write and close).

For each collective write operation for client-directed I/O the data are written to the first level cache on the aggregators, while the server-directed I/O transfers the data to the AHPIOS server in the second level cache. As a consequence in most cases client-directed I/O hides better the write latency to the applications. However, when the final flushing is done the server-directed I/O outperforms client-directed I/O due to the fact that the file close is done right after the last write. Therefore, there is no overlapping with communication or computation for this last step.

BTIO closes the file after writing and then reads the data for verification. For client-directed I/O the write data are propagated from the first level cache to the second level cache, but a copy of the data is kept in the first-level cache. Therefore, the client-directed collective I/O routines read the data from the first level cache. This explains the better results for collective reads when using client-directed I/O in most of the cases.

We note that AHPIOS client-directed I/O and server-directed I/O significantly outperform the write and read operations of the other three methods in all cases. In general client-directed

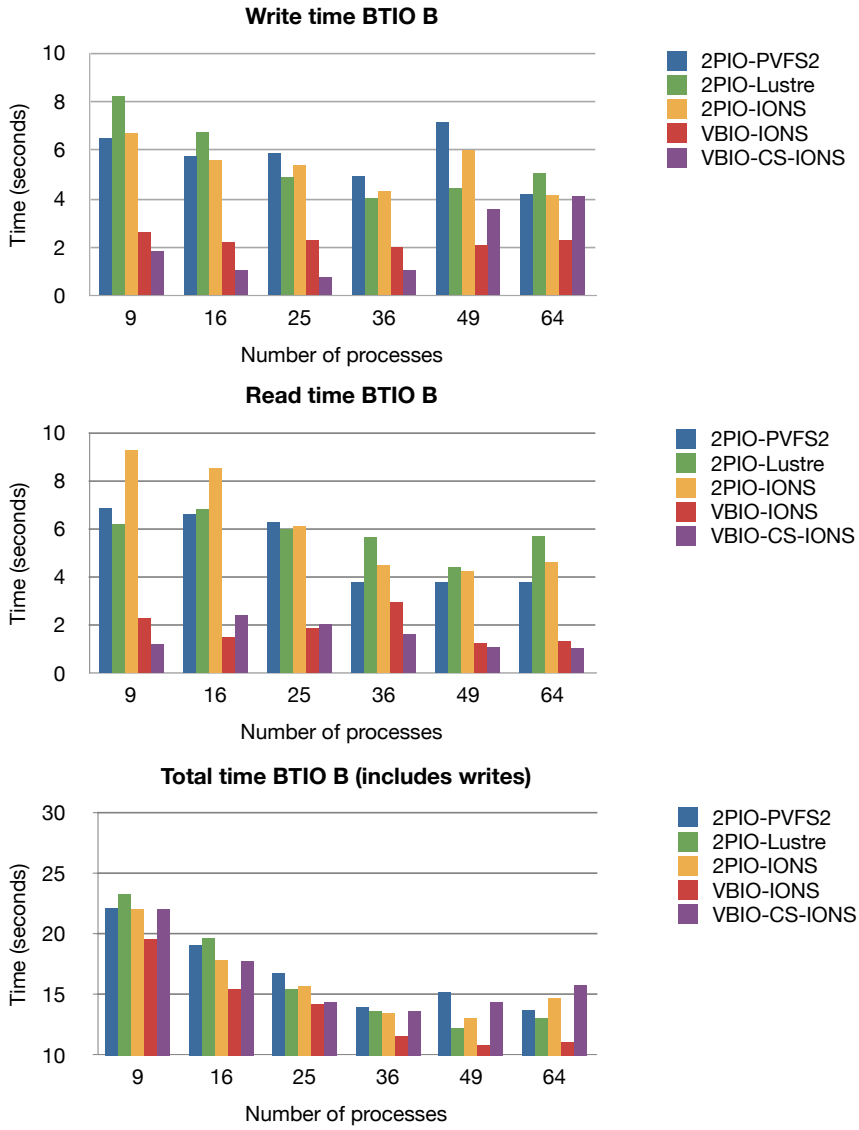

Figure 6.8: *AHPIOS. BTIO class B measurements. We compare five different solutions for parallel I/O access: ROMIO two-phase I/O over PVFS2 (2PIO-PVFS2), ROMIO two-phase I/O over Lustre (2PIO-Lustre), ROMIO two-phase I/O over AHPIOS (2PIO-IONS) and the two AHPIOS-based solutions: server-directed I/O (VBIO-IONS) and client-directed I/O (VBIO-CS-IONS).*

I/O hides the latency better in most cases, because data are written to the clients and then staged through both cache layers. However, when the file close time is included, server-directed I/O performs the best in all cases, because data were already transferred to the AHPIOS servers asynchronously. In particular for class B, client-directed I/O hides best the latency of file writes up to 36 processes, but does not scale good due to the fact that aggregators, as communication hubs, become overloaded: each aggregator receives data from all MPI processes, shuffle them and transfer them to all AHPIOS servers. In turn, server-directed I/O performs better for increasing contention, as the communication is reduced. For class C, as the communication volume is larger, the better performance of client-directed I/O over server-directed I/O is marginal for small number of processes. As for class B, file writes of server-directed I/O scale better. The file reads

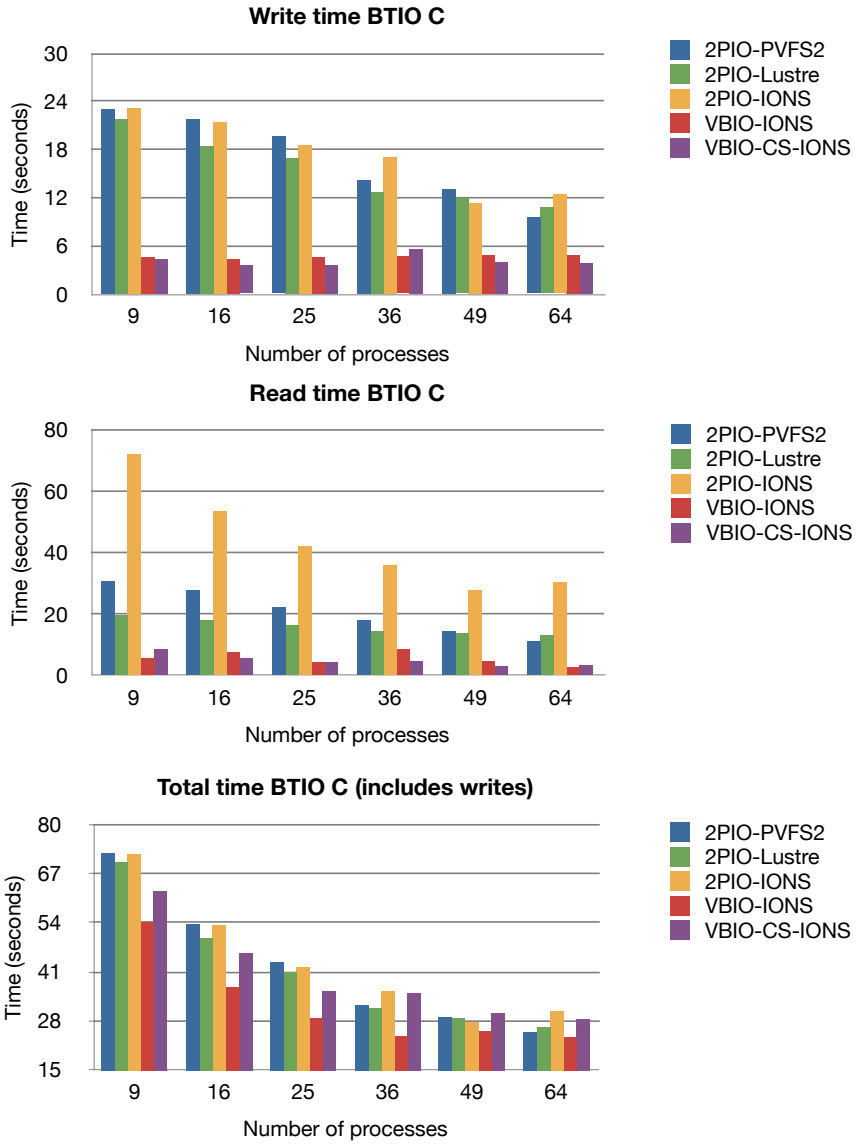

Figure 6.9: *AHPIOS. BTIO class C measurements. We compare five different solutions for parallel I/O access: ROMIO two-phase I/O over PVFS2 (2PIO-PVFS2), ROMIO two-phase I/O over Lustre (2PIO-Lustre), ROMIO two-phase I/O over AHPIOS (2PIO-IONS) and the two AHPIOS-based solutions: server-directed I/O (VBIO-IONS) and client-directed I/O (VBIO-CS-IONS).*

perform substantial better for both server-directed I/O and client-directed I/O than for two-phase I/O, as the data is read from the collective cache levels: level 1 for client-directed I/O and level 2 for server-directed I/O. The performance is similar for both methods, as the communication volume and contention levels are roughly the same. In terms of total time reported by the application, which includes the write-time and the time to completely flush the data to the AHPIOS servers (read time is not included), server-directed I/O performs best in all cases, as the client-directed I/O incurs the cost of flushing of remainder data of level 1 cache.

There are additional reasons explaining these results. In two-phase I/O, data are in general transferred twice over the network: first, for data aggregation at compute nodes, and second, for

accessing the file system. In client-directed I/O, the aggregation is done at the AHPIOS server, i.e. close to the storage. If the storage is locally available, the second communication operation is spared.

Additionally, the view-based I/O technique significantly reduces the size of the metadata sent over network. The MPI data types are sent in a compact form to the AHPIOS servers at view declaration. This data type transfer is done only once and data types can be reused by subsequent I/O operations. In contrast two-phase I/O requires the lists of (file offset, length) tuples to be sent to the aggregators at each file operation.

MPI Tile I/O benchmark

MPI Tile I/O benchmark [MPI] evaluates the performance of MPI-IO library and file-system implementation under a non-contiguous access workload. The benchmark logically divides a data file into a dense two-dimensional set of tiles. The number of tiles along rows (nr_x) and columns (nr_y) and the size of each tile in the x and y dimensions(sz_x and sz_y) are specified as input parameters. We have chosen these values such that for any number of processes the total amount of data accesses is 1GByte and the access granularity is 4KBytes. Table 6.4 lists the values of these parameters. The size of an element is 1 byte.

Table 6.4: *Parameters of MPI Tile I/O benchmark.*

Process count	sz_x (KBytes)	sz_y (KBytes)	nr_x	nr_y
4	4	64	4	4
8	4	32	4	4
16	4	16	4	4
32	4	8	8	4
64	4	4	8	8

The performance results are plotted in Figure 6.10. Client-directed I/O (VBIO-CS-IONS) scales very well with the problem size. In this case the cache scales with the number of processes, because the clients put in common parts of their memory. 2PIO-Lustre (two-phase I/O over Lustre) performs comparably to the others for small numbers of processes, but does not scale, even though the two-phase I/O aggregators cache the file blocks in their local memory. Client-directed I/O does not scale beyond 8 processes, which represents the same number as the AHPIOS servers employed in this experiment. AHPIOS performs much better than all other approaches and scales linearly. A larger number of AHPIOS servers would contribute to AHPIOS scalability, as seen in Section 6.2.2.

MPI implementation evaluation

AHPIOS is fully implemented in MPI. MPI offers a powerful paradigm for programming on distributed memory systems in terms of programming facility, portability and performance. We found very convenient the employment of both point-to-point and collective operations. Both blocking and non-blocking point-to-point routines are straightforward to use and with a minimal management overhead (when compared with TCP/IP sockets for instance). The collective

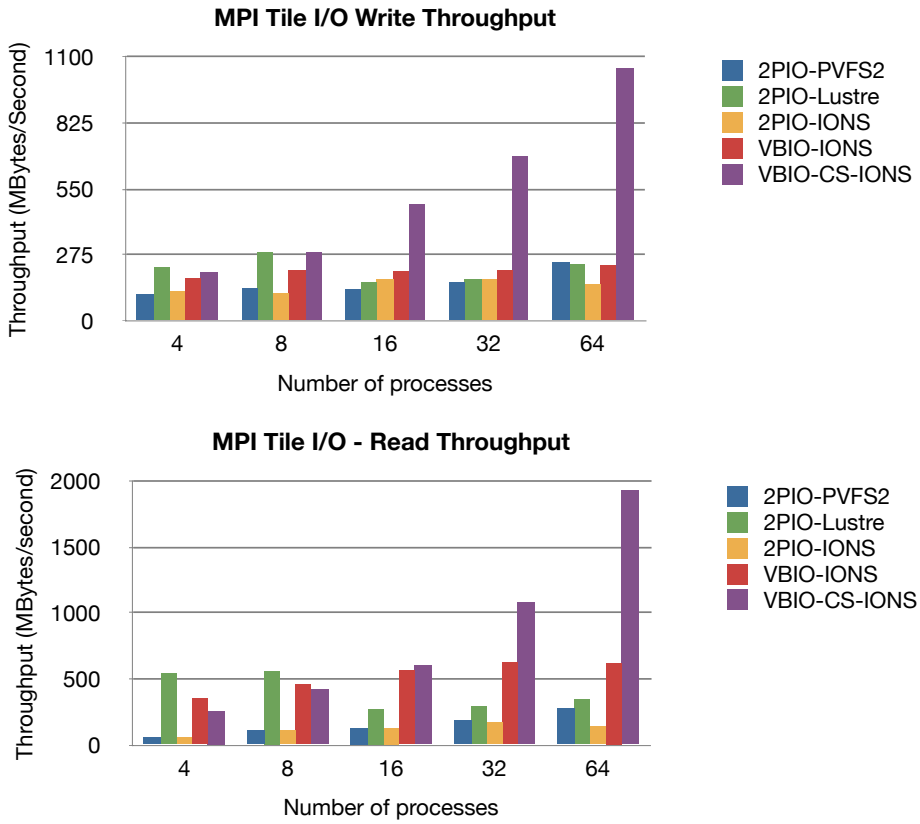

Figure 6.10: *AHPIOS. MPI Tile I/O throughput. For any number of processes the total amount of data accesses is 1GByte and the access granularity is 4KBytes.*

synchronization routines such as `MPI_Barrier` offer the possibility of a rapid (conservative) implementation and facilitate the debugging.

However, one of the most important advantages is the portability, the AHPIOS system can run unmodified on all the systems that have installed an MPI library.

One problematic aspect we found was the error mechanisms implemented in MPI. In some cases an MPI implementation must provide an explicit handling of errors, for instance for propagating an error from a faulting process. A global automatic exception mechanism for MPI would significantly increase the productivity of implementing in MPI.

In order to better understand the employment of MPI in the solutions presented, we traced the BTIO application by using the performance profiling library MPE [GKL95]. We profiled the execution for 9 processes, because it was one of the cases for which the performance differed significantly in all runs and because the size of the generated trace is small enough (around 4 MBytes). Tables 6.5 and 6.6 show the number of communication and synchronization calls performed by BTIO for I/O purposes. Table 6.5 shows the number of MPI point-to-point calls of all processes, while table 6.6 gives the number of collective calls. Each collective call is counted once for a group of processes that perform it.

Note that AHPIOS server-directed I/O implementation employs only blocking point-to-point communication (`MPI_Send`, `MPI_Recv`), while the other implementations are using both blocking and non-blocking point-to-point operations (`MPI_Isend`,`MPI_Irecv`, `MPI_Waitall`). Therefore,

Table 6.5: *The count of MPI point-to-point communication operations for NP=9 processes.*

MPI call	2PIO-PVFS2	2PIO-Lustre	2PIO-IONS	Server-directed (VBIO-IONS)	Client-directed (VBIO-CS-IONS)
MPI_Send	0	0	4660	4660	4660
MPI_Recv	0	0	4660	4660	5830
MPI_Isend	5670	5760	5760	0	5553
MPI_Irecv	5670	5760	5760	0	4383
MPI_Waitall	855	855	855	0	396

Table 6.6: *The count of MPI collective communication and synchronization operations for NP=9 processes.*

MPI call	2PIO-PVFS2	2PIO-Lustre	2PIO-IONS	Server-directed (VBIO-IONS)	Client-directed (VBIO-CS-IONS)
MPI_Bcast	60	60	60	0	0
MPI_Barrier	22	25	25	7	154
MPI_Allreduce	22	25	25	7	154
MPI_Alltoall	60	60	60	0	0
MPI_Allgather	20	20	20	0	0

the performance of both AHPIOS server-directed and client-directed can be improved by a subsequent implementation with non-blocking operations.

For AHPIOS the numbers from the tables include the communication to the distributed storage, i.e. the communication overhead for providing a transparent parallel I/O access to files. In the case of Lustre and PVFS these operations are performed by using internal communication protocols and can not be directly compared with the MPI routines. This fact explains why client directed I/O and 2PIO-IONS generate a significant higher number of MPI messages. However, even under these conditions, the communication is lower for server-directed I/O than for 2PIO-PVFS2 and 2PIO-Lustre.

The number of collective communication and synchronization operations performed by the two-phase I/O implementation is considerably higher. The two-phase I/O collective buffering is done at compute nodes, which need to perform expensive all to all operations in the shuffle phase: in order to get the list of file offset-length pairs and the data. The 2PIO-PVFS2, 2PIO-Lustre and 2PIO-IONS solutions performed a similar number of operations (2PIO-IONS and 2PIO-Lustre used 3 more barrier operations).

Discussion

The performance results show that the tight integration of application and storage system together with the asynchronous data staging strategy and the cooperative caching bring a substantial benefit over the traditional solutions.

The two-level distributed file cache scales with the number of processes at the first level and with the number of storage resources at the second level. The strategy of asynchronous

data staging between the caching levels hides the latency of file accesses from the applications. The experiment demonstrates that AHPIOS offers a substantial performance benefit over the traditional MPI-IO solutions on both PVFS and Lustre parallel file systems.

6.2.3 GPFS-based I/O

The goal of this subsection is to present an evaluation of our GPFS-based parallel I/O architecture in clusters.

The measurements from this evaluation were performed on NCSA, which is part of the Teragrid open scientific infrastructure[Ter]. NCSA's cluster consists of 887 IA-64 IBM nodes: 256 nodes with dual 1.3 GHz Intel Itanium2 processors and 631 nodes with dual 1.5 GHz Intel Itanium2 processors. The benchmarks ran on the 1.5GHZ Itanium2 procesors, equipped with 4GBytes of memory per node. GPFS was configured with 42 I/O servers and 512KBytes file block size. The communication interconnect is Myrinet-2000, with advertised MPIMX latency of 2.6μs-3.2μs and throughput of 247MBytes/second. This cluster uses a Linux kernel version 2.4.21 and IBM GPFS version 3.1.0. The employed MPI distribution was MPICH2 1.0.5. The communication protocol of MPICH2 was TCP/IP on top of Myrinet. We ran all experiments with one process per compute node.

In the following experiment we compare five different solutions for GPFS-based parallel I/O access: ROMIO two-phase I/O over GPFS (2PIO), the original view-based I/O (VBIO), view-based I/O with client-side write-back (VBIO-CS), and the two GPFS-based solutions: data-shipping based I/O (DSIO) and prefetching MAR-based I/O (MARPIO). Given that MAR hint allows granularities of prefetching of at most one file block, we have set the size of the collective buffer for 2PIO and VBIO to the size of the file block, i.e 512KBytes.

File write evaluation

First, we used gpfsPerf in order to evaluate the raw performance of GPFS (outside ROMIO). Figure 6.11 shows a comparison between aggregate throughputs for data-shipping and POSIX write operations for record sizes from 32KBytes to 1MBytes. As expected, data-shipping achieves higher throughputs than POSIX for access granularities lower than the file system page (512KBytes) due to fact that local caching is disabled and there is no locking overhead. Additionally, the throughput scales well with the number of application nodes, especially for small record sizes (32KBytes).

POSIX write operations perform better for granularities larger than the file block size, as GPFS locking granularity is a file block. Therefore, the local caching benefit is larger than the locking cost. For 16 processes and a record size of 512KBytes there is a spike in the POSIX performance corresponding to the case of the record size being exactly the same as the file block size.

We conclude from this first experiment that data-shipping may bring substantial benefits only for access granularities smaller than a file block. For file block granularities POSIX offers the best solution. Therefore, in the remainder of this section we concentrate on the evaluation of small granularity accesses, which are typical for an important class of parallel scientific applications.

This type of small granularity access is characteristic for FLASH I/O and BTIO benchmarks, for which we evaluated the performance of collective file write operations. In our evaluation, we

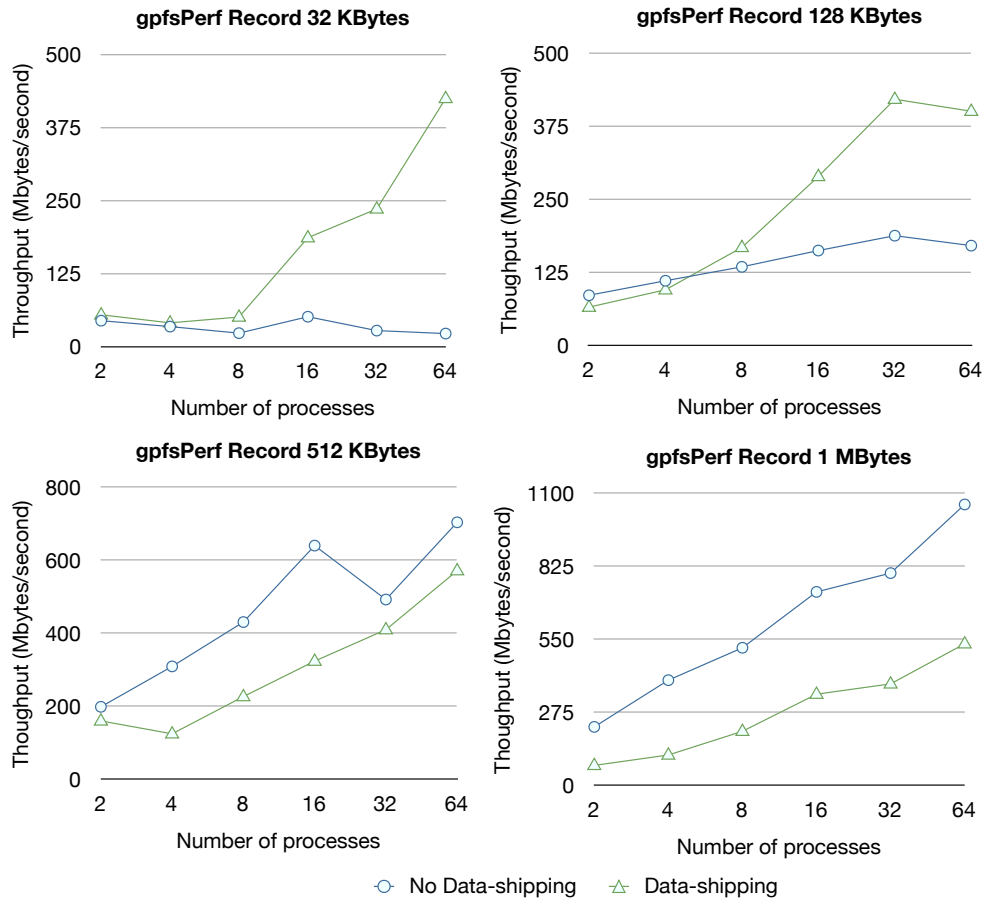

Figure 6.11: *GPFS-based I/O. gpfsPerf performance for strided pattern access. Comparison between aggregate throughputs for data-shipping and POSIX write operations for record sizes from 32KBytes to 1MBytes. Data-shipping may bring substantial benefits only for access granularities smaller than a file block.*

compare two-phase collective I/O (2PIO), data-shipping based I/O (DSIO), original view-based I/O (VBIO) and view-based I/O with client-side write-back (VBIO-CS). In the case of FLASH I/O, which is a stress I/O test including only I/O phases, there is no write-back benefit for view-based I/O. Therefore, we show only the results for VBIO. In these experiments view-based I/O cache size was set to 64MBytes.

Figure 6.12 plots the aggregate throughput in FLASH I/O. We use 2 to 64 processes and a 16x16x16 data set. In this case each MPI process writes approximately 60MBytes (checkpoint file) and 8MBytes (centered and corner data files). We note that the VBIO performs better for large files. The checkpoint file write performance results for 64 compute nodes show that VBIO attains a 20% improvement compared to 2PIO and DSIO. Center and right graphs show that, for small files DSIO performs best, followed in the most of the cases by VBIO and DSIO.

Figure 6.13 shows a breakdown of the BTIO time into file write time, file close time and computation times for a class B execution, which writes a file of around 1.6GBytes of data. First, it can be noticed that VBIO-CS outperforms the other methods in all cases. The compute time is similar for a given number of compute nodes. The better performance of VBIO-CS comes from the overlapping of write time and close time with the compute time. As it can be noticed, the

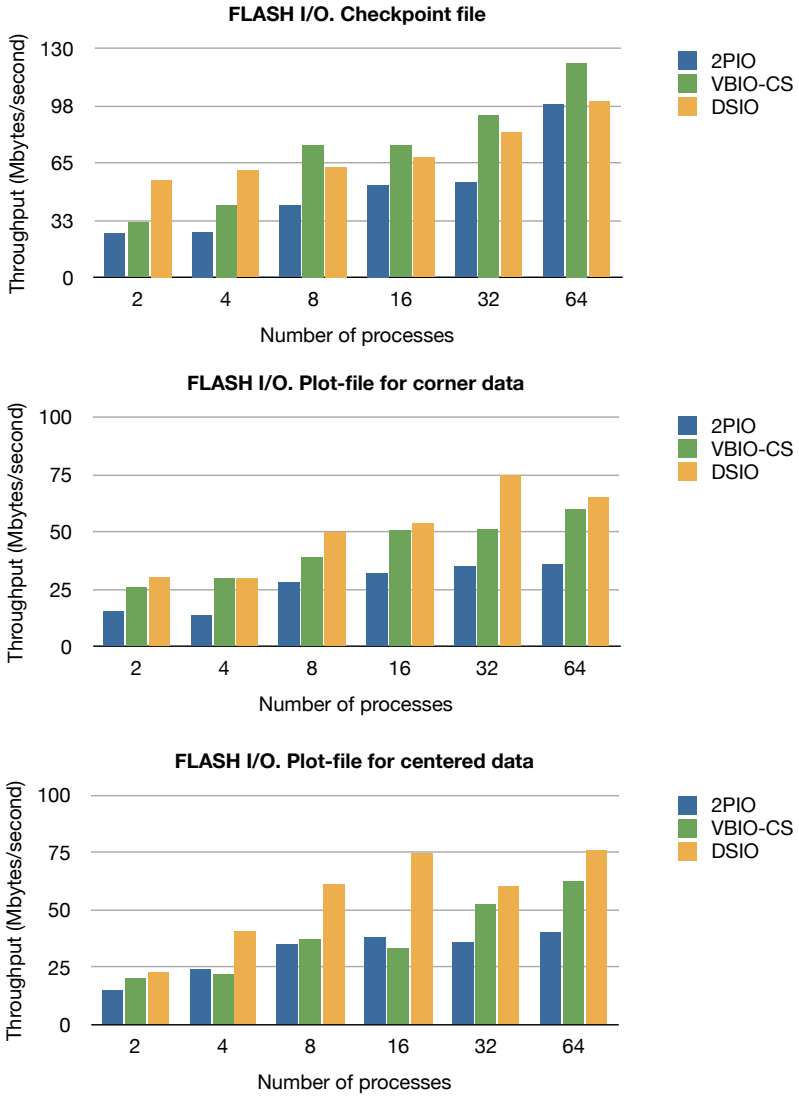

Figure 6.12: *GPFS-based I/O. FLASH I/O performance of two-phase I/O (2PIO), DS-based I/O (DSIO), and view-based I/O (VBIO). Each MPI process writes approximately 60MBytes (checkpoint file) and 8MBytes (centered and corner data files), resulting a file of 3.8GBytes.*

latency perceived by the application is smaller for these components of the total time. 2PIO does not scale with the number of compute nodes, as the granularity of access of BTIO decreases and, therefore, the shuffle time of the first phase increases, as it will be confirmed also by the collective read results. DSIO does not scale either, the time to access the file decreases only slightly with the increase of the number of processes.

File read evaluation

In this subsection we evaluate the performance of GPFS-based file read operations.

As in the case of file writes, we first evaluated the file read operations outside the ROMIO implementation. We run the gpfsPerf benchmark with strided read access patterns for a number

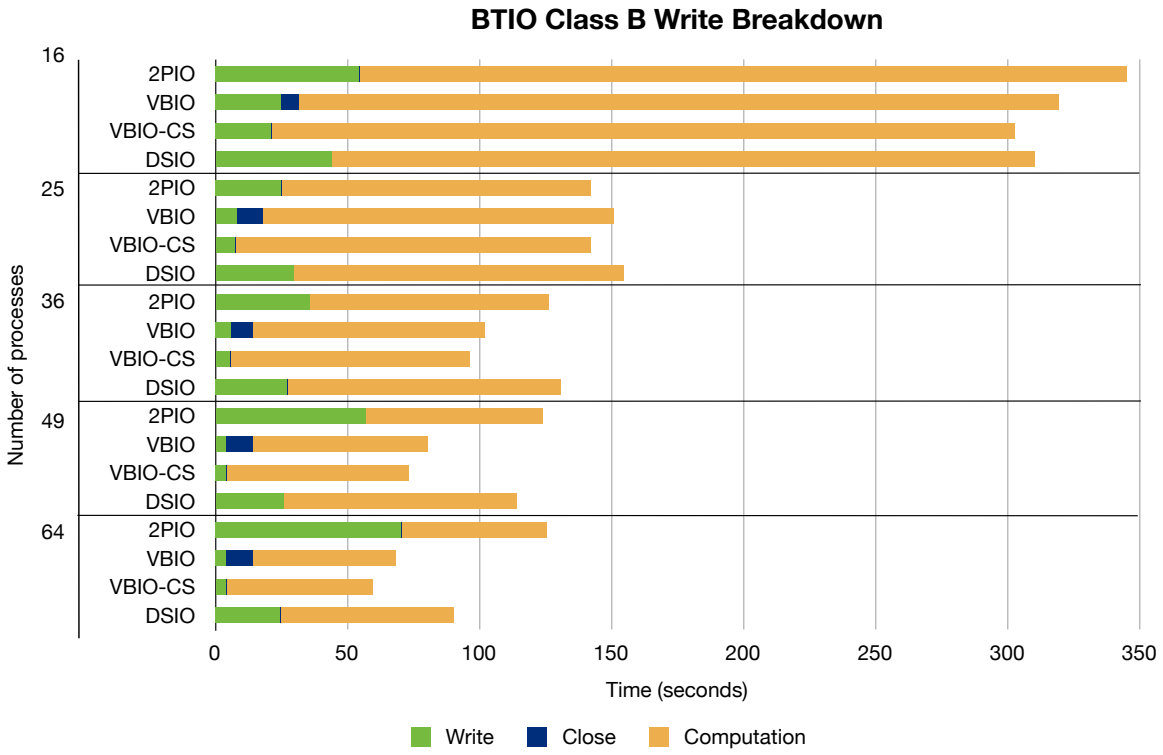

Figure 6.13: *GPFS-based I/O. BTIO class B file write performance. Figure shows a breakdown of the BTIO time into file write time, file close time and computation times for a class B execution, in which BTIO writes a file of around 1.6GBytes of data.*

of MPI processes varying from 2 to 64. In one setup we have fixed the record size, from 32KBytes to 512KBytes, and varied the number of processes from 2 to 64. In the second setup we use a fixed number of processes (16 and 64 nodes) and vary the record size from 1KByte to 512KBytes.

We evaluate the effect of MAR prefetching hint on the read access performance. Even though GPFS recognizes strided access patters and performs prefetching accordingly, Figure 6.15 demonstrates that the employment of MAR hint provides substantial better performance for record sizes smaller than 32KBytes.

For SimParIO the nodes read interleaved strided access patterns with record sizes of 8KBytes, 16KBytes and 32KBytes. We chose these values, as the Figures 6.14 and 6.16 indicate that the activated MAR hint provides high throughput for small granularities. The application performs 20 iterations, in each of which all processes read with strided access pattern with stride *64 x record size*. For record sizes of 8KBytes, MARPIO offers best throughputs for low numbers of processes. The performance degrades for higher number of processes due to high contention of all processes for all blocks: for 64 processes all processes read from the same file block (8KBytes x 64). The performance of MARPIO improves significantly for 16KBytes and 32KBytes records sizes, as the contention decreases. VBIO-CS shows comparable results for 32 and 64 processes and 32KBytes record sizes.

Figure 6.17 shows the total read time for BTIO class B. For the collective implementations (2PIO, VBIO, VBIO-CS, and MARPIO) we present a breakdown into the time dedicated to I/O related communication from the shuffle phase and the file access time. VBIO-CS perfoms best

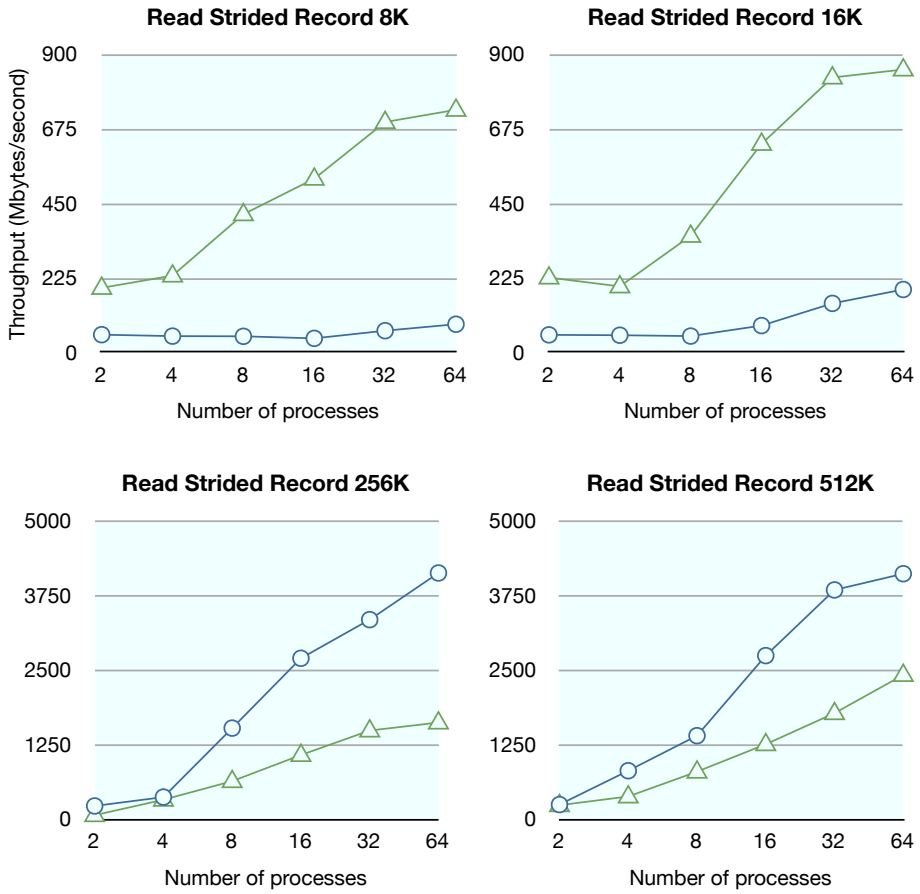

Figure 6.14: *GPFS-based I/O. gpfsPerf file read performance for a strided access pattern. Fixed the record size, 32KBytes to 512KBytes, and varied the number of processes from 2 to 64.*

only for 16 processes. In general, VBIO-CS partially hides the latency of file accesses seen by VBIO, by overlapping the file accesses with the communication from shuffle phase. MARPIO performs considerably better for larger number of processes. For 64 processes MARPIO provides a 100% improvement over second-best performing method (VBIO-CS). The explanation for this behaviour is that as the number of processes increases, the file access granularity decreases, and, as shown in Figure 6.16, MAR hints provide more performance benefits for small granularity accesses. In turn, in the case of VBIO-CS the access granularity is one file block, i.e. 512KBytes.

Discussion

The performance evaluation shows that the combination of collective strategies with overlapping of computation, communication, and I/O, may bring a substantial performance benefit for access patterns common for parallel scientific applications. We conclude from our experiments that data-shipping I/O performs best only for access granularities smaller than a file block. We also conclude that hiding the latency of file writes brings a substantial benefit for the applications. This benefit appears to increase with the decrease of the granularity of access. For read operations, even when they do not take advantage of collective buffers, view-based collective I/O performance is better than the one for two-phase I/O in all cases.

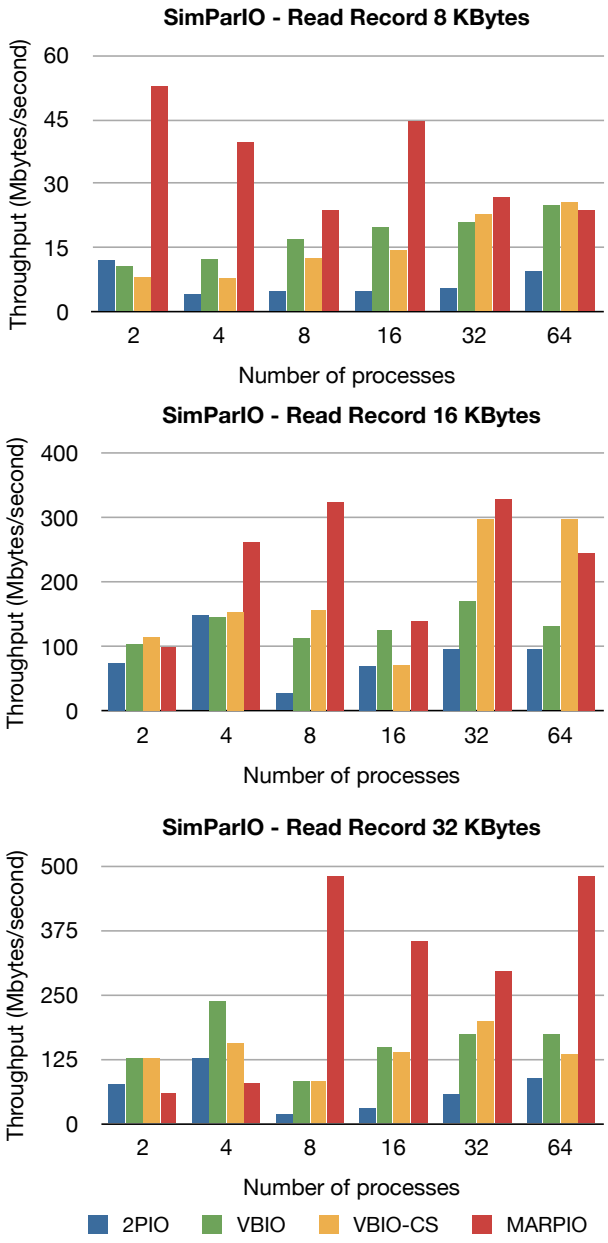

Figure 6.15: *GPFS-based I/O. SimParIO file read performance for strided access pattern. The application performs 20 iterations, in each of which all processes read with strided access pattern with stride* 64 x record size. *For record sizes of 8KBytes, MARPIO offers best throughputs for low numbers of processes.*

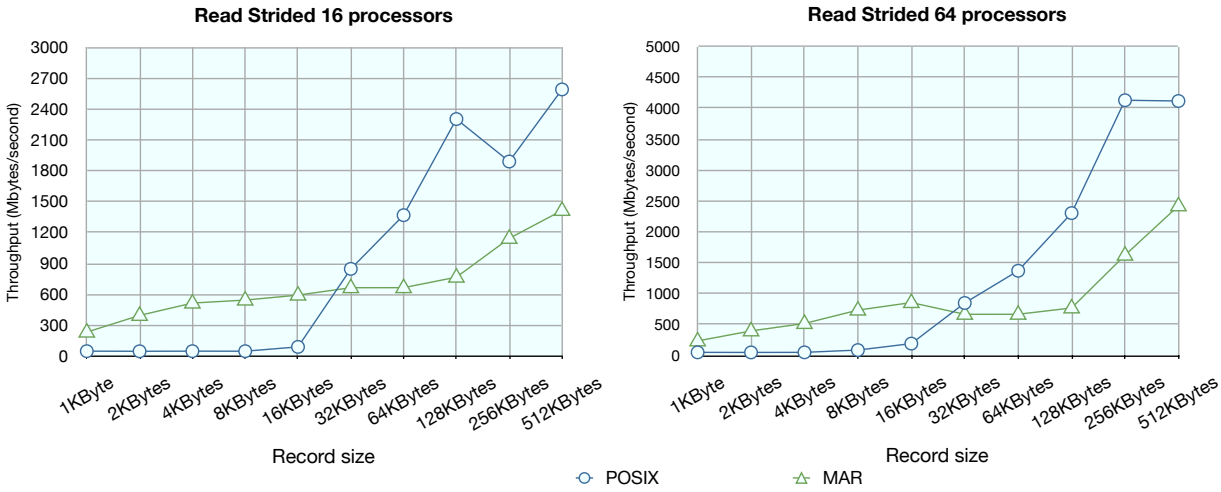

Figure 6.16: *GPFS-based I/O. gpfsPerf file read performance for a strided access pattern. Fixed number of processes (16 and 64 nodes) and vary the record size from 1KBytes to 512KBytes.*

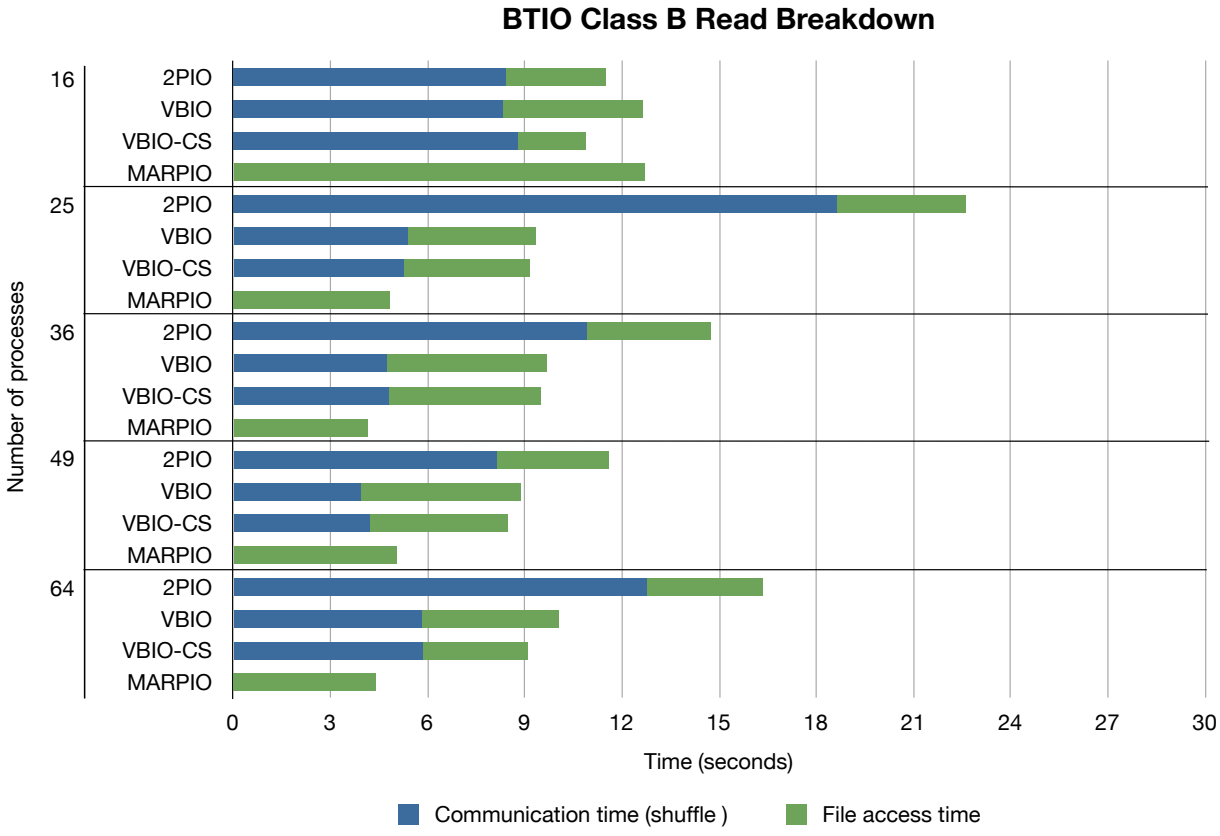

Figure 6.17: *GPFS-based I/O. BTIO class B file read performance for four collective implementations: two-phase I/O (2PIO), view-based I/O (VBIO), view-based I/O with client-side prefetching (VBIO-CS), and MAR-based prefetching I/O (MARPIO).*

6.3 Evaluation on supercomputers

The evaluation targets to demonstrate that our data staging techniques achieves high performance on a multi-threading system such as Blue Gene/L and Blue Gene/P. Our evaluation targets to answer the following questions: What is the benefit of employing multiple level file caching on compute nodes and I/O nodes? Does the use of the torus network for file access optimization pay off? Which asynchronous policies are suitable for data staging (pipelining)? What coordination is needed? What are good ratios/sizes of file caches on different levels of the hierarchy?

6.3.1 Blue Gene/L

In this section, we describe the performance evaluation of our parallel I/O architecture on Blue Gene/L systems.

The experiments presented in this section have been performed on the Blue Gene/L system from Argonne National Laboratory. The system has 1024 dual-core 700 MHz PowerPC 440 processors with 512MBytes of RAM. Level 1 caches are not hardware coherent, therefore, Blue Gene/L's compute nodes do not support multi-threading. Data is transferred between compute nodes and I/O nodes over the a global tree network with a bandwidth of *2.8Gb/link*. Each pset of 32 compute nodes is served by one I/O node. All 32 I/O nodes are interconnected to 14 storage nodes though a Gigabit Ethernet interface. The storage nodes provide mass storage for the BlueGene/L system. Each storage node contains a ServeRAID 6i+ SCSI RAID Controller, which connects to six internal 146.8GBytes 10K SCSI HDDs for a total of 880GBytes raw storage per server or 14TBytes total raw storage (11.7TBytes usable). Requests to storage nodes are served by 4 xSeries 346 servers with dual 3.4 GHz Xeon processors, 4GBytes RAM. All the experiments were run in coprocessor mode (one MPI process per node).

Contiguous access

We have run the SimParIO benchmark in order to evaluate the performance of contiguous access, using the caching only on I/O nodes. Caching on compute nodes was not evaluated given the fact that Blue Gene/L did not support thread on compute nodes, making impossible the asynchronous transfers of data to I/O nodes.

The number of alternating phases was 20. The maximum file size produced by 512 processes and record size of 1MByte was 10GBytes. The compute nodes do not use any caching. We compare four cases: IBM's CIOD-based solution (CIOD), view-based I/O without cache (VBIO), ZOID with cache and zero-time compute phase (VBIO-IONS with cache 0s) and ZOID with cache with a 2 seconds compute phase (VB-IONS with cache 2s). The I/O node thread starts flushing data to the file system, when more than half of the cache contains dirty blocks (50% high water mark).

In one setup we have fixed the record size (we report two cases: 128KBytes and 1MBytes) and varied the number of processes from 32 (one pset) to 512 (16 psets). In the second setup we use a fixed number of processes (we report two cases: 64 and 512 processes) and vary the record size from 1KBytes to 1MBytes. Figures 6.18, 6.19, 6.20, and 6.21 display the results: in the upper row the aggregate throughput of all nodes and in the lower row the file close times. The file close times are relevant because the remaining dirty blocks of I/O node cache are flushed to the file

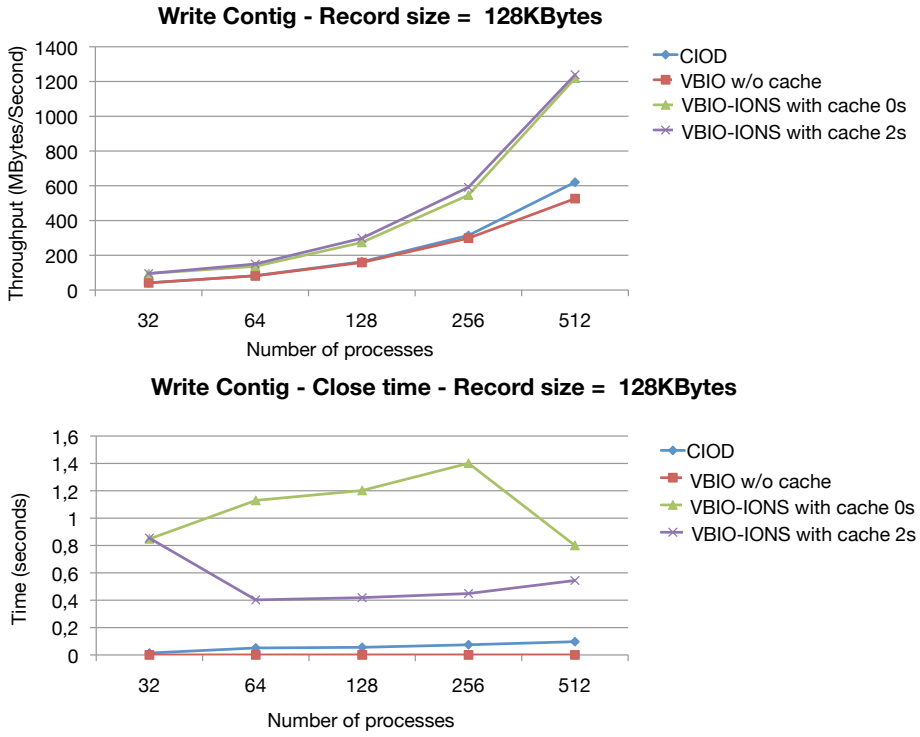

Figure 6.18: *BG/L. Performance of contiguous access for a fixed record size of 128KBytes. The graph shows how the aggregate throughput scales when the compute nodes grow.*

system at close time.

As expected, we can notice that VBIO-IONS solutions caching file data at I/O nodes significantly outperform the CIOD and VBIO-IONS without cache. The close times of caching solutions is larger than for non-caching solutions. However, the close time pays off when compared with the benefit in terms of aggregate throughput. The close time is smaller for the 2-seconds compute phase, as the cache is asynchronously flushed to the file system while the computing proceeds.

The aggregate throughput is especially large for small records for both 64 processes and 512 processes. For small records the file system latencies are large, therefore, the effects of latency hiding have a substantial impact.

BTIO benchmark

We have run the BTIO benchmark with three different variants of collective I/O techniques: two-phase I/O (the original collective I/O implementation from ROMIO and CIOD), view-based I/O with no cache on the I/O nodes (VBIO) and view-based I/O with a cache of 128MBytes on the I/O nodes (VBIO-IONS). All collective I/O implementation were using the ZOIDFS module. All nodes acted as aggregators.

Figures 6.22 and 6.23 show the total write time and total application time for the class C (writing around 6.4GBytes of data). The upper row depicts the total write time and the lower row the overall application times. For 64 processes the graph shows an increase in access times when the compute node cache becomes full. In the case of 256 processes, the file fits completely in the cache of the compute nodes and no change in performance was noted. It can be noticed

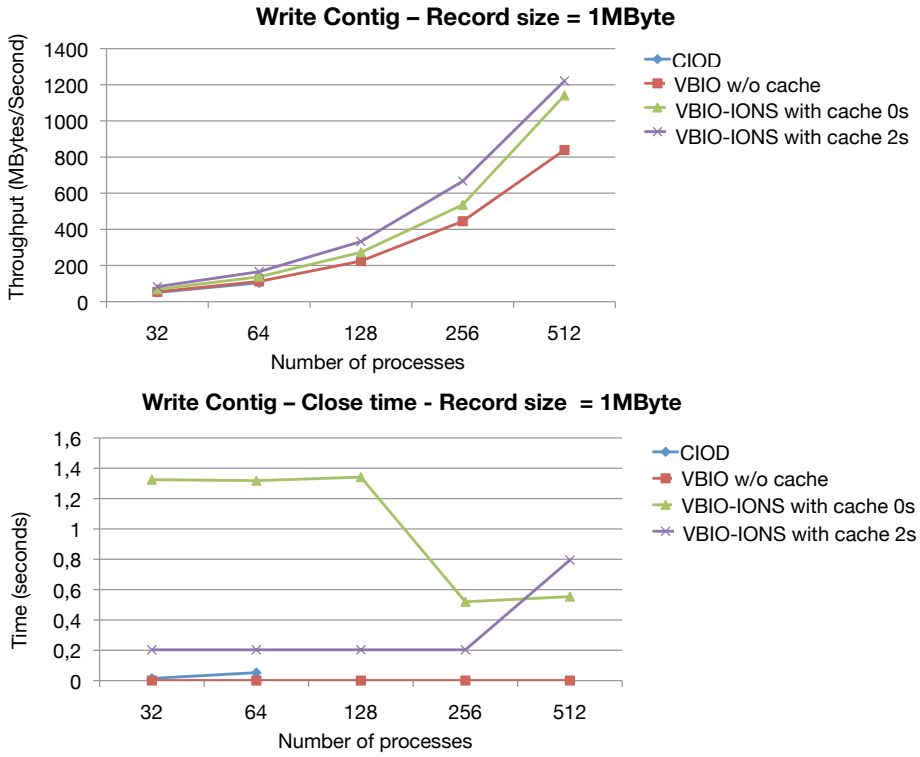

Figure 6.19: *BG/L. Performance of contiguous access for a fixed record size of 1MByte. The graph shows how the aggregate throughput scales when the compute nodes grow.*

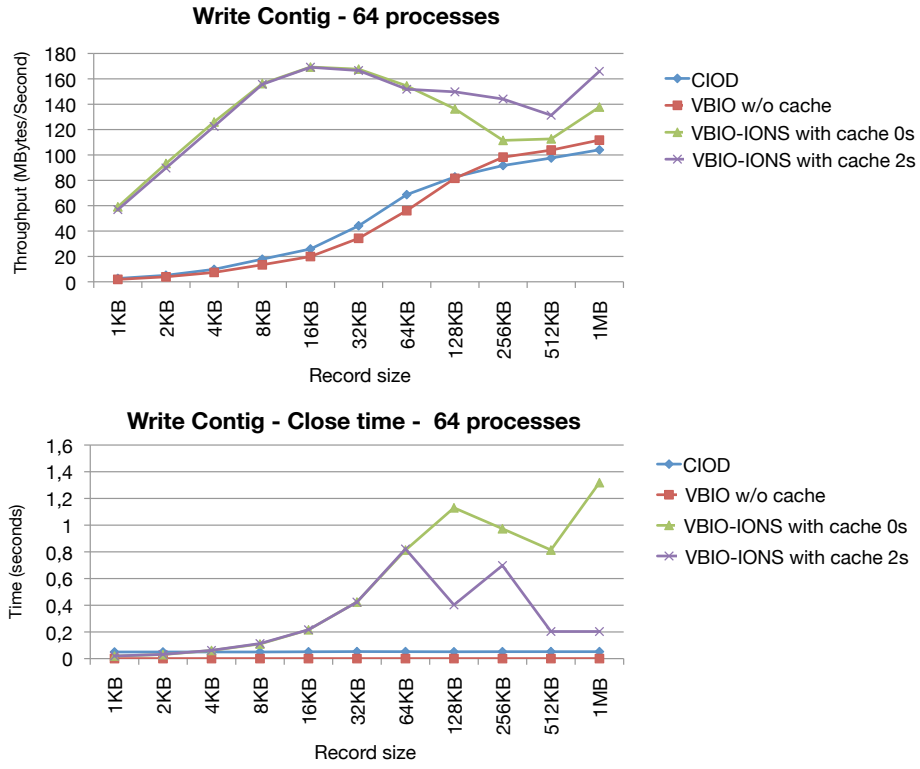

Figure 6.20: *BG/L. Performance of contiguous access for different record sizes and 64 processes.*

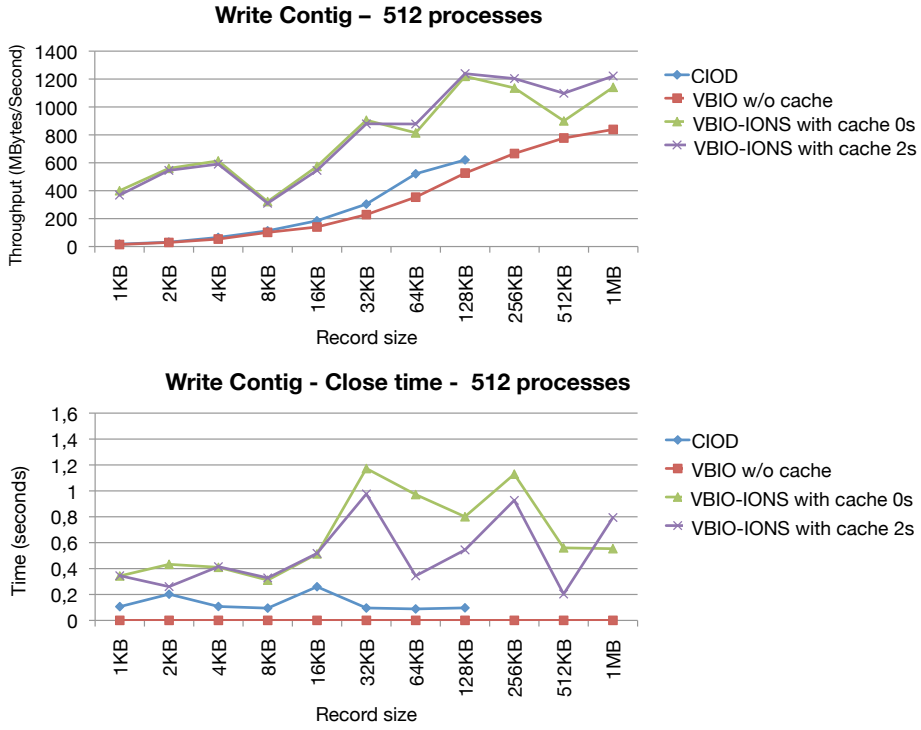

Figure 6.21: *BG/L. Performance of contiguous access for different record sizes and 512 processes.*

also that two phase I/O has a higher cost in the first access and that, subsequently, flushes the collective buffer at each file access.

Figure 6.24 displays the write times of all of the 40 I/O steps. The file write time seen by the application is significantly lower for both cached and uncached cases of view-based I/O than for two-phase I/O. This is due to the caching at the compute nodes. However, the improvement in total application time is lower, because this compute node cache can not be flushed asynchronously due to the lack of threads support on Blue Gene/L architecture.

6.3.2 Blue Gene/P

The experiments have been performed on the Blue Gene/P system from Argonne National Laboratory. The system has 1024 quad-core 850 MHz PowerPC 450 processors with 2GBytes of RAM. All the experiments were run in Symmetric Multiprocessor mode (SMP), in which a compute node executes one MPI process per node with up to four threads per process. The PVFS 2.8.1 file system at Argonne consists of four servers. The PVFS files were striped over all four servers with a stripe size of 4MBytes.

Table 6.7 summarizes the configurable parameters employed in the experiments.

File system and network performance

In this section we perform an initial evaluation of Blue Gene system in order to obtain the optimal file system and network configuration parameters for torus network, tree network, and

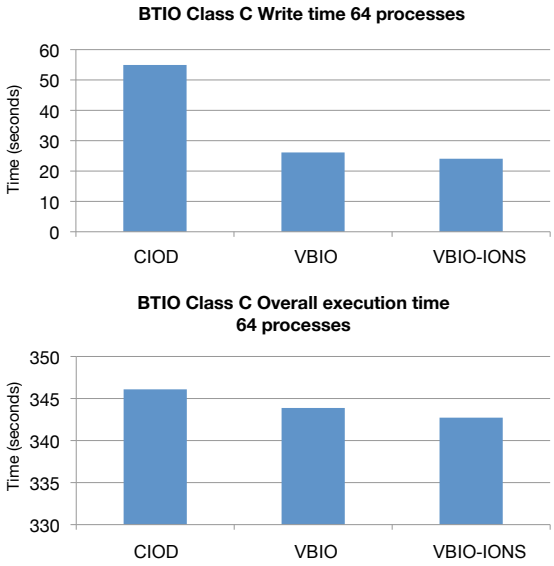

Figure 6.22: *BG/L. BTIO class C times for 64 processes. Comparative of two-phase I/O over CIOD (CIOD), view-based I/O with no cache on the I/O nodes (VBIO) and view-based I/O with a cache of 128MBytes on the I/O nodes (VBIO-IONS).*

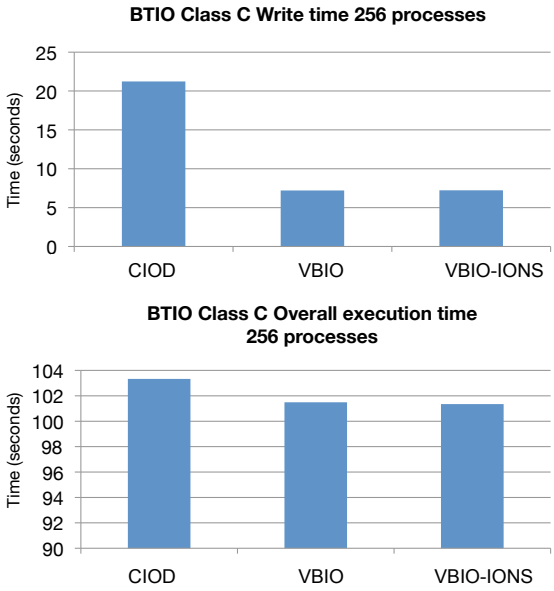

Figure 6.23: *BG/L. BTIO class C times for 256 processes. Comparative of two-phase I/O over CIOD (CIOD), view-based I/O with no cache on the I/O nodes (VBIO) and view-based I/O with a cache of 128MBytes on the I/O nodes (VBIO-IONS).*

file system.

Figure 6.27 plots PVFS file write and read throughputs for access from one I/O node. For both write and read operations, the best performing access size for both write and read operations is 4MBytes, corresponding the stride size of PVFS. Figures 6.25 and 6.26 plot the network throughput for message sizes on the torus and tree networks, respectively. For both networks, we note that 4MBytes is a suitable message size, equal to the selected message size for

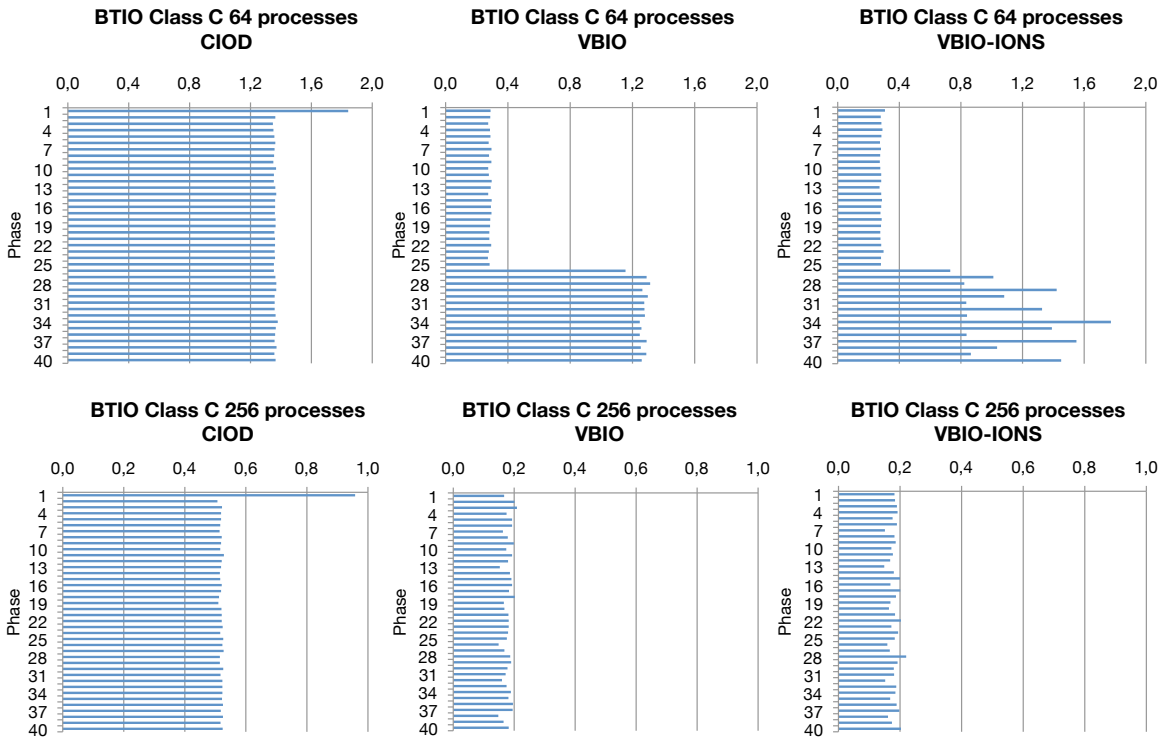

Figure 6.24: *BG/L. Times for all steps of BTIO class C. Figure displays the write times of all of the 40 I/O steps.*

file accesses. In the following experiments we will set up the network and file system buffer sizes in 4MBytes.

Scalability

The following experiments target to evaluate and analyze the scalability of our solution with number of aggregators and compute nodes.

Aggregators. We have run the SimParIO benchmark in order to evaluate the file access scalability using caching only on compute nodes. Compute nodes concurrently write and read contiguously and non-contiguously a file stored over a pset for different number of aggregators. In all experiments, the I/O node file cache is disabled. The time used to calculate the aggregate throughput includes the time to flush the caches on file close.

Figure 6.28 shows the aggregate file write when varying the number of aggregators, from 1 to 64 (full pset). The number of alternating phases was 40. The maximum file size produced by 64 processes acceding records of size of 2MBytes was 5.2GBytes. The figure shows the results for high water mark values between 0% and 25%. The maximum client-side file cache size is 8GBytes for 64 aggregators (128Mbytes per compute node). For 64 aggregators, the file fits entirely in cache. We compare four cases: IBM's CIOD-based solution (CIOD), client-side file cache with different compute phases (VBIO-CS 0ms, VBIO-CS 500ms, and VBIO-CS 1000ms). In all cases, we observe that CIOD base solution performs worst, especially when the number of aggregators increases. We observed that increasing the number of agrregators, increases the performance,

Table 6.7: *Experiment parameters on BG/P.*

Figure	Operation	Compute nodes	Aggregators per pset	I/O nodes	Cache sizes		Access pattern	File size
					Clients	I/O nodes		
6.28	Write	64	1 to 64	1	128MB	-	Contiguous	5.2Gbytes
6.29	Write	64	4 to 64	1	128MB	-	Strided	10Gbytes
6.30	Write	256	4 to 64	4	128MB	-	Strided	40Gbytes
6.31	Read	64	4 to 64	1	128MB	-	Strided	10Gbytes
6.32	Write	64 to 512	64	1 to 8	128MB	512MB	Strided	up to 0.5TBytes
6.33	Read	64 to 512	64	1 to 8	128MB	512MB	Strided	up to 0.5TBytes
6.34	Write	64	64	1	0 to 128MB	0 to 512MB	Contiguous	10GBytes
6.35	Write	64	64	1	128MB	512MB	Strided	10GBytes
6.36	Write	256	256	4	128MB	512MB	Strided	40GBytes
6.37	Write	64	64	1	128MB	512MB	Strided	10GBytes
6.38	Read	64	64	1	128MB	512MB	Contiguous	10GBytes
6.39	Read	64	64	1	128MB	512MB	Contiguous	10GBytes
6.40	Write	64 and 256	64	1 and 4	128MB	512MB	Strided	1.6GBytes
6.41	Read	64	64	1	128MB	512MB	Strided	1.6GBytes
6.42	Read	64	64	1	128MB	512MB	Strided	1.6GBytes

especially when the file fits completely in the cache (64 aggregators). The best case corresponds to a value of high water mark of 12.5%. The performance increases in the presence of computation starting from a high water mark of 12.5%.

Figure 6.29 shows the aggregate write throughput for compute phases of 0ms and 500ms, when varying the number of aggregators and using caching only on compute nodes. Compute nodes perform 40 iterations. In each iteration, the file access pattern is strided with a record size of 64KBytes and a stride size of 4Mbytes. The file size produced by 64 compute nodes was 10GBytes. The maximum client-side file cache size is 8GBytes for 64 aggregators. The file does not fit in client-side file cache in either of the cases. We compare four cases: IBM's CIOD-based solution (CIOD) and our client-side file cache solution with different compute phases (VBIO-CS 0ms, VBIO-CS 500ms, and VBIO-CS 1000ms). In absence of computation, we note that the increase of the number of aggregators does not affect the performance CIOD or VBIO-CS. The graph shows that without computation, VBIO-CS offers a 70% improvement. For 500ms and 1000ms compute phases, VBIO-CS scales with the number of aggregators, especially for 1000ms compute phases.

Figure 6.30 plots the aggregate write throughput of 256 processes accessing a non-contiguous file of 40GBytes. The figure shows the aggregate write throughput for compute phases of 0ms and 500ms, when varying the number of aggregators and using caching only on compute nodes. The x-axis indicates the number of aggregators for each pset. It is important to note that the aggregators are deployed in a balanced manner over 4 psets. Compute nodes perform 40 iterations. In each iteration, the file access pattern is strided with a record size of 64KBytes and a stride

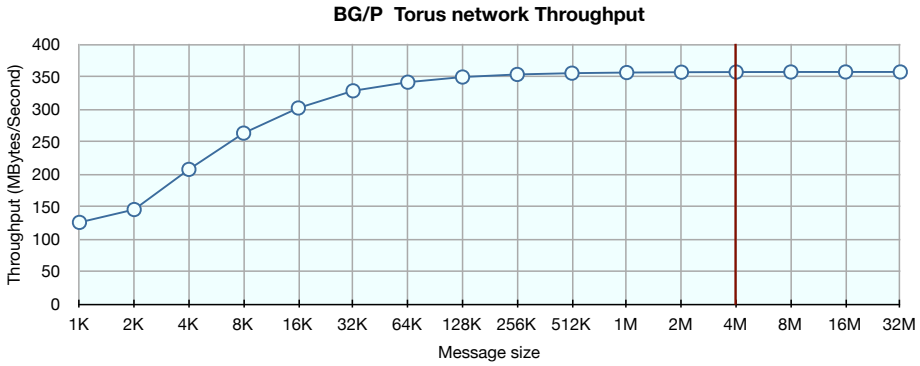

Figure 6.25: *BG/P. Torus network throughput for different message sizes.*

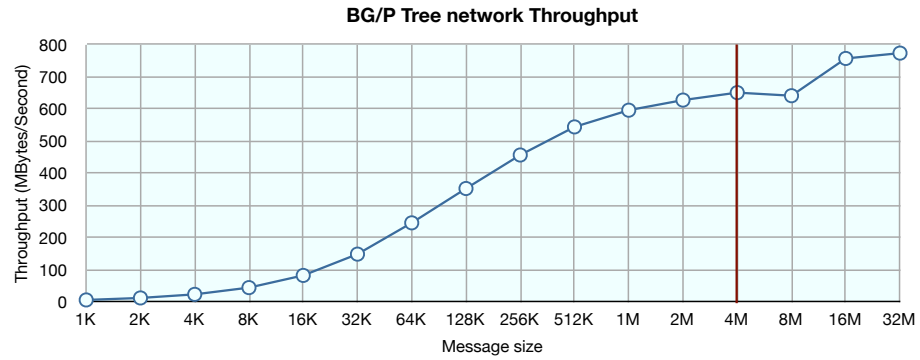

Figure 6.26: *BG/P. Tree network throughput for different message sizes.*

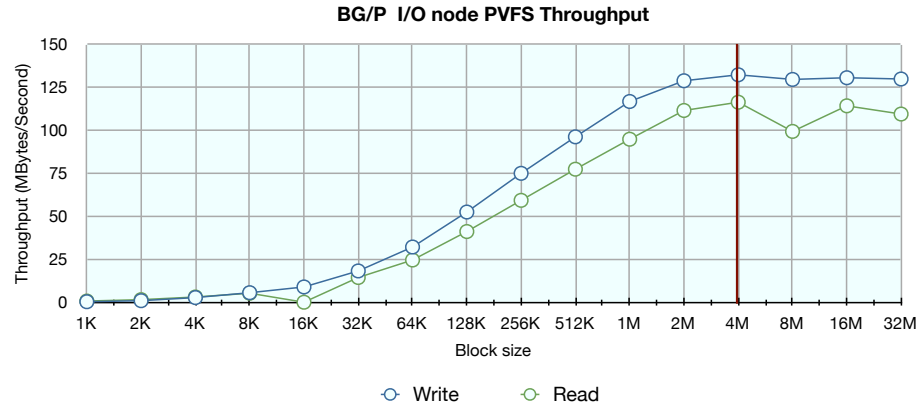

Figure 6.27: *BG/P. PVFS file write and read throughput on an single I/O node.*

size of 4Mbytes. The file size produced by 256 processes was 40GBytes. The maximum client-side file cache size is 32GBytes for 256 aggregators. The file does not fit entirely in the client-side file cache in either of the cases. The high water mark for client-side cache was 12.5%. We observe that even without compute phases, VBIO-CS scales significantly when all compute nodes act as aggregators.

Figure 6.31 plots the aggregate read throughput for compute phases of 0ms and 500ms,

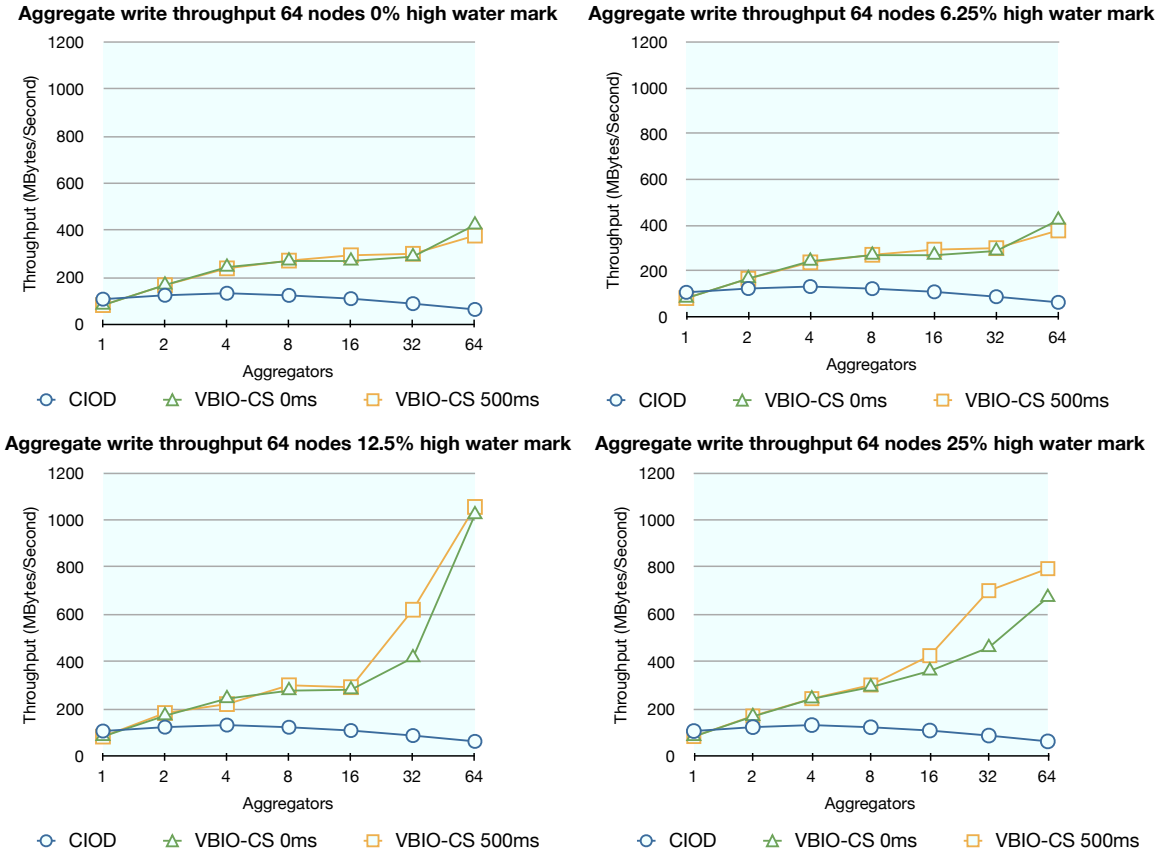

Figure 6.28: *BG/P. Effect of number of aggregators on the aggregate file write throughput for a pset of 64 processes, 0% to 25% high water mark for client-side cache, 0% high water mark for I/O node-side cache, 0ms and 500ms compute phase. The benchmark generates a file of 10Gbytes of contiguous data.*

when varying the number of aggregators. In the upper row of the figure, compute nodes read a contiguous file. The maximum file size accessed by 64 processes and a record size of 2MBytes was 5.2GBytes. The maximum client-side file cache size is 8GBytes for 64 aggregators. For 64 aggregators, the file fits entirely in cache. In the lower row of the figure, compute nodes read a non-contiguous file. The file size produced by 64 processes was 10GBytes. The maximum client-side file cache size is 8GBytes for 64 aggregators, in any case, the file does not fit in client-side file cache. For both contiguous and non-contiguous access patterns, the application performs 40 iterations, and each compute node prefetches up to 32 blocks in the cache. For contiguous and non-contiguous data accesses, and in the absence of computation, we note that the increase of aggregators does not affect CIOD or VBIO-CS read throughput. However, for 500ms compute phases, VBIO-CS outperforms CIOD in all cases. The prefetching efficiency increments with the number of aggregators, due to fact that the aggregated cache size increases and a higher number of buffers is prefetched. For 500 compute phases, the read accesses have a higher margin to be overlapped with computation.

File size and compute nodes number. This evaluation aims to test the solution scalability in terms of file size and number of compute nodes. Figures 6.32 and 6.33 plot the results of running SimParIO benchmark with 64 to 512 compute nodes. The size of client-side cache on a

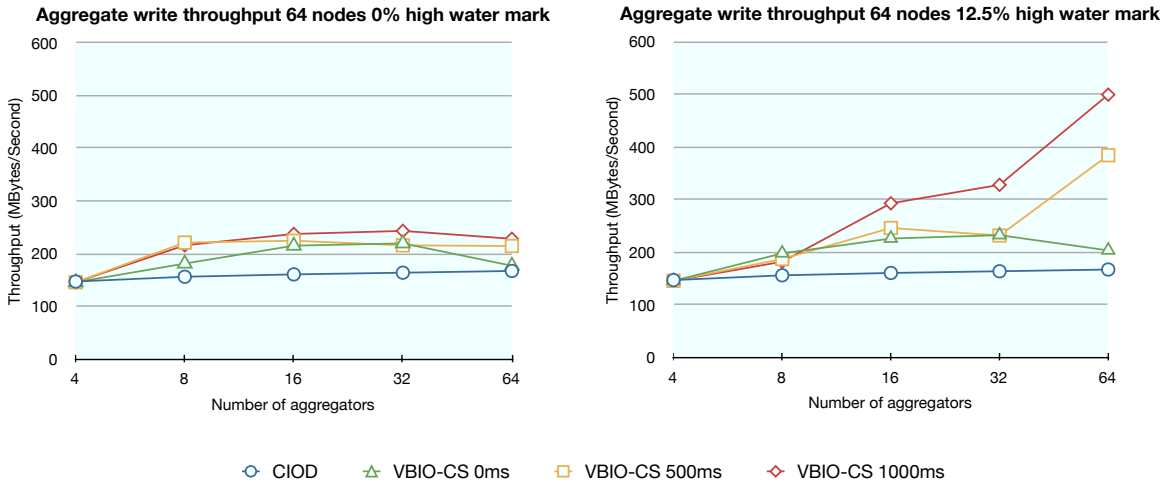

Figure 6.29: *BG/P. Effect of number of aggregators on the aggregate file write throughput for 64 processes, 0% and 12.5% high water mark for client-side cache, 0% high water mark for I/O node-side cache, 0ms to 1000ms compute phases. The benchmark generates a file of 10Gbytes and accesses non-contiguous data.*

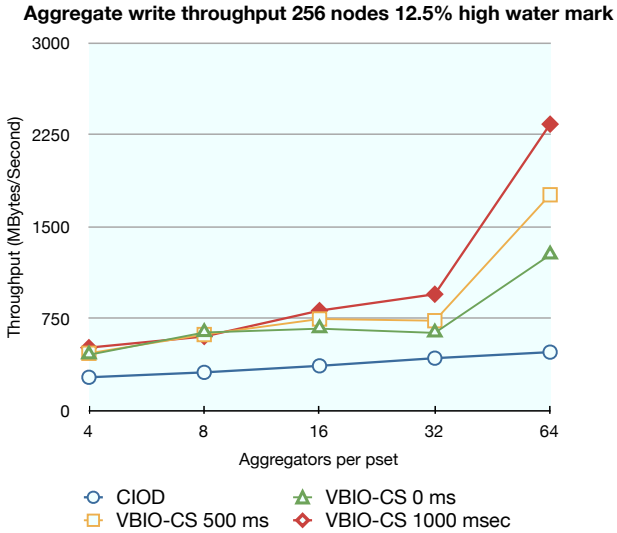

Figure 6.30: *BG/P. Effect of number of aggregators on the aggregate file write throughput for four psets, 256 processes, 12.5% high water mark for client-side cache, 0% high water mark for I/O node-side cache, 0ms to 1000ms compute phases. The benchmark generates a file of 40Gbytes and accesses non-contiguous data.*

compute node is 128MBytes. All the compute nodes cache data (act as aggregators). The size of the I/O node cache is 512MBytes. The block size is 4MBytes for both client and I/O node file caches. The high water mark for client-side cache was 12.5%, and for the I/O node-side cache 12.5%. Compute nodes perform 40 iterations. In each iteration, the file access pattern is strided with a record size of 64KBytes and a stride size of 4MBytes. The maximum file size produced by 512 processes was 0.5TBytes. We evaluate the file writes of the SimParIO benchmark for three different setups: two-phase I/O over IBM solution (CIOD), view-based I/O with client-side caching (VBIO-CS), and view-based I/O with both client-side and I/O node-side caching

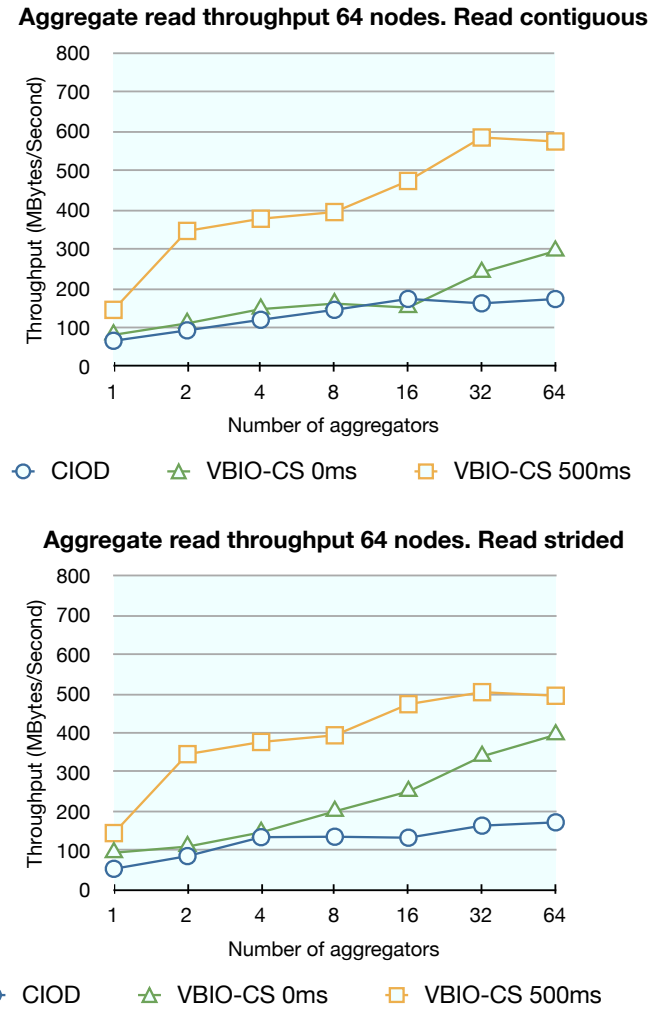

Figure 6.31: *BG/P. Effect of number of aggregators on the aggregate file read throughput for 64 processes, 32 read ahead buffers, 0ms to 500ms compute phases. The application reads a file of 10Gbytes and accesses both contiguous (upper row) and non-contiguous data (lower row).*

(VBIO-CS-IONS). The graphs show that file access performance scales well with the number of compute nodes for both read and write operations. The performance obtained is higher when both cache levels are employed and in presence of computation.

File cache sizes

This experiment targets to evaluate and analyze the dependence of the file write-back performance on the sizes of the file caches on the compute node and I/O node. The experiment was run on 64 compute nodes inside a pset. The high water mark for client-side cache was 6.25%, and for the I/O node-side cache 6.25%.

The size of client-side cache on a compute node was varied from 0MBytes (no caching) to 128MBytes. The size of I/O node-side cache was varied from 0MBytes (no caching) to 512MBytes. All the compute nodes cache data (act as aggregators). The file block size is 4MBytes for both

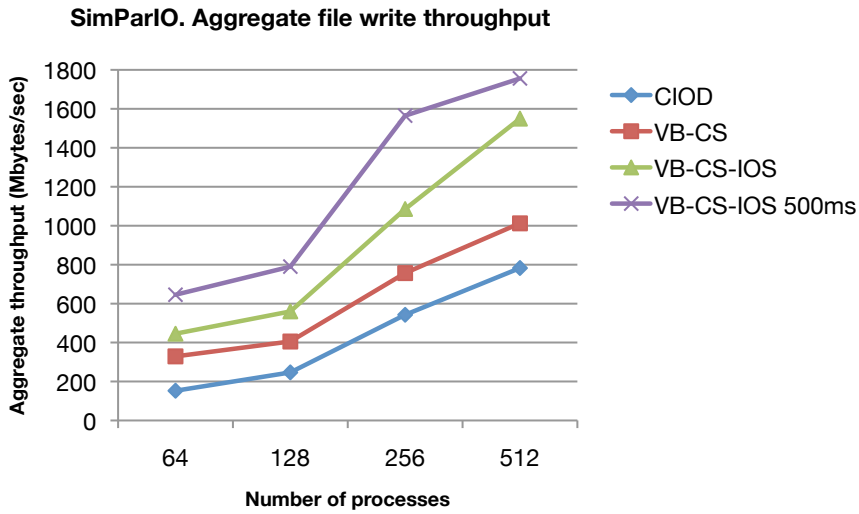

Figure 6.32: *BG/P. Effect of number of compute nodes on the aggregate file write throughput for 64 to 512 processes, 0ms to 500ms compute phases.*

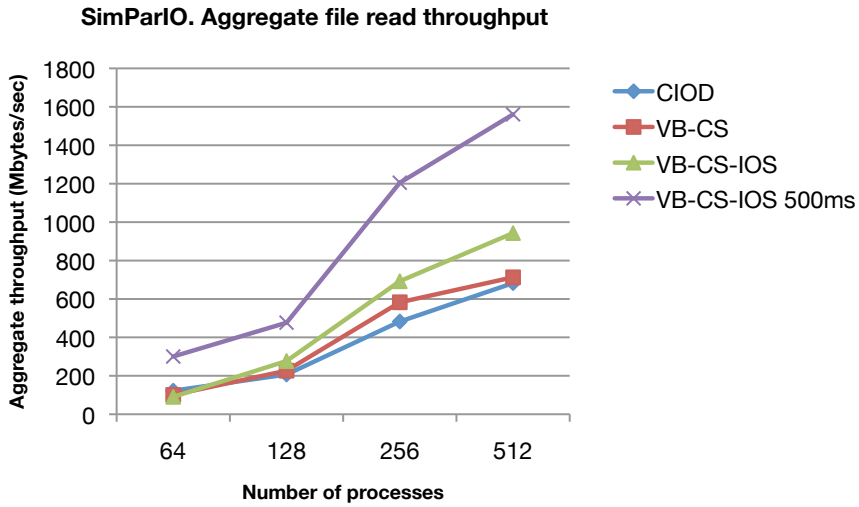

Figure 6.33: *BG/P. Effect of number of compute nodes on the aggregate file read throughput for 64 to 512 processes, 0ms to 500ms compute phases.*

client and I/O node file caches. SimParIO was configured to write a total of 10GBytes, i.e. each process repeatedly writes a record of 4MBytes in 40 phases.

Figure 6.35 shows the aggregate file write for compute phases of 0ms (upper row) and 1000ms (lowe row). When computing the throughput, the time to close the file is included.

The graphs show that the client-side caches bring a substantial performance improvement. This improvement is almost independent of the size of the I/O caches. The best results are obtained for client-side caches of 64MBytes and I/O node-side caches of 512MBytes. The ratio between the best client-side cache size and I/O node-side caches size is 64, corresponding to the number of compute nodes in the pset and to the number of aggregators. This result suggests the optimal size of the I/O node cache to be equal to the sum of the client-side caches in the corresponding pset. Further increases of this cache appear even to worsen the performance. This

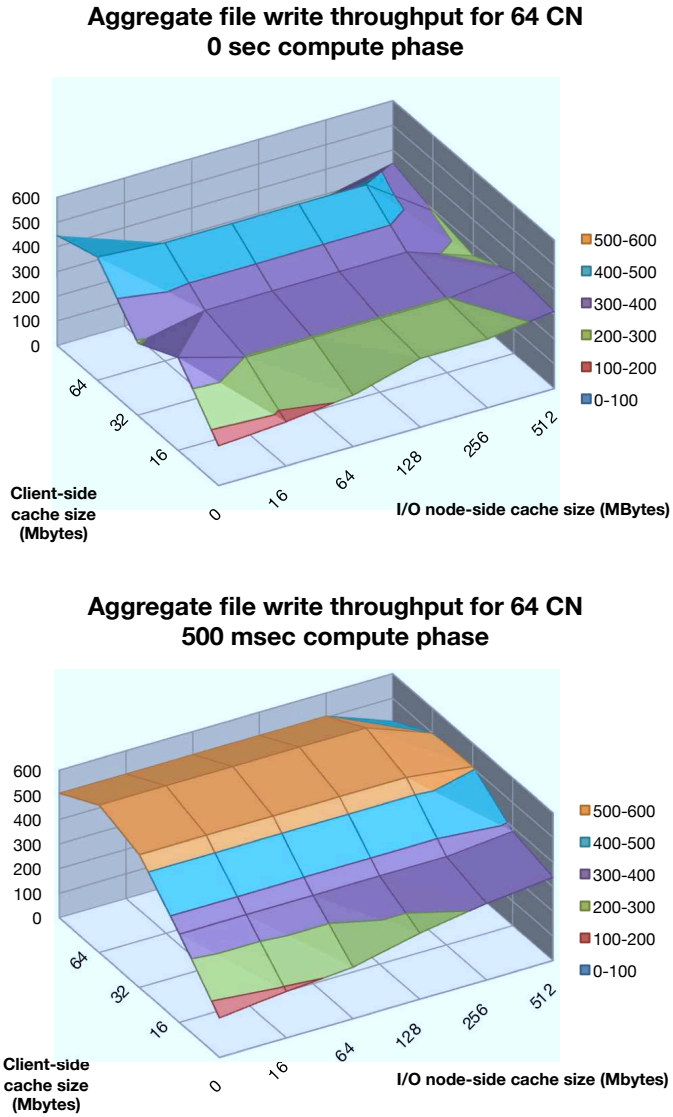

Figure 6.34: *BG/P. Effect of file cache sizes on the aggregate file write throughput for a pset of 64 processes, 6.25% high water mark for client-side cache, 6.25% high water mark for I/O node-side cache, 0ms and 1000ms compute phase. The ratio between the best client-side cache size and I/O node-side caches size is 64, corresponding to the number of compute nodes in the pset and to the number of aggregators.*

could be explained by the fact that a larger cache may take a longer time to be flushed when the file is closed.

Expectedly, the size of the cache closer to the application (client-side cache) appears to influence the performance in a stronger way than a remoter cache (I/O node-side cache). The comparison of the two graphs for 0ms and 1000ms shows that a better potential to overlap computation brings only a marginal performance benefit for client-side caches larger or equal to 16MBytes.

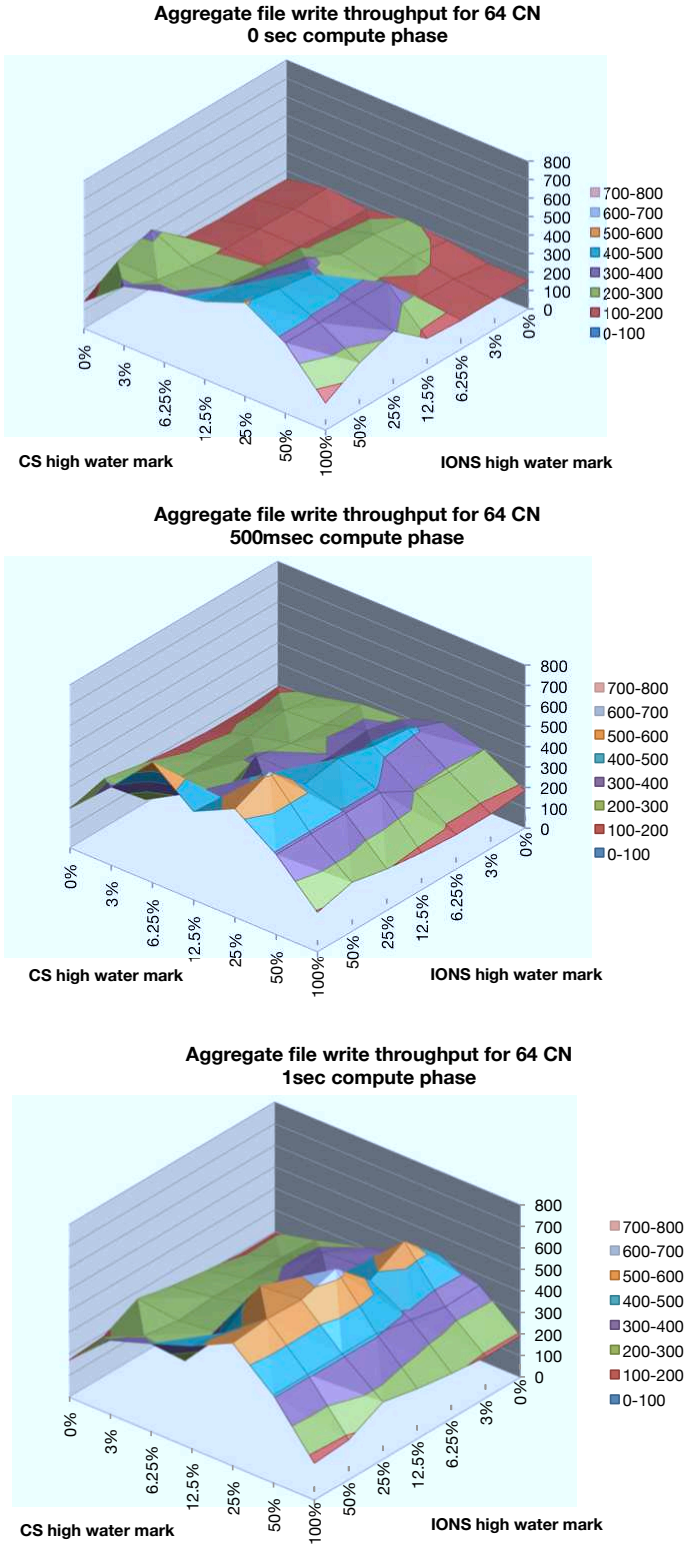

Figure 6.35: *BG/P. File write performance for 64 processes, client-side cache size of 128MBytes, I/O node-side cache size of 512MBytes, variable high water mark and 0, 500, and 1000ms compute phases. The time used to calculate the aggregate throughput includes the time to flush the caches on file close. The throughput increases with the decrease of the high water mark on the I/O nodes and is highest for client-side high water mark of 6.25% and 12.5%. In presence of computation the throughput increases for small client-side high water marks.*

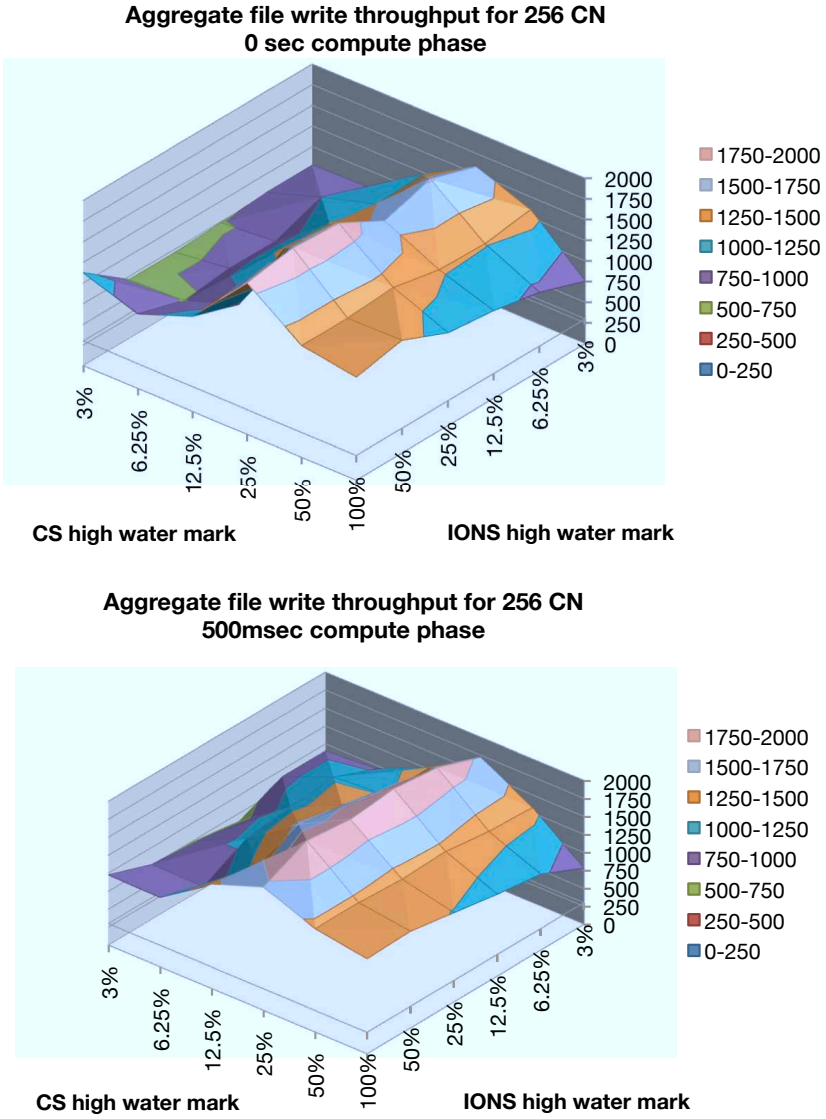

Figure 6.36: *BG/P. File write performance for 256 processes, client-side cache size of 128MBytes, I/O node-side cache size of 512MBytes, variable high water mark and 0 and 500ms compute phases. The time used to calculate the aggregate throughput includes the time to flush the caches on file close.The throughput increases with the decrease of the high water mark on the I/O nodes and is highest for client-side high water mark of 12.5% and 25%.*

File write performance

This experiment targets to evaluate and analyze the dependence of the file write-back performance on the high-water marks of the file caches on the compute node and I/O node. The size of client-side cache on a compute node is 128MBytes. All the compute nodes cache data (act as aggregators). The size of the I/O node cache is 512MBytes. The block size in both client and I/O node-side caches is 4MBytes. SimParIO was run with 64 and 256 processes (one process per compute node) and was configured to write a total of 10GBytes for 64 processes and 40GBytes for 256 processes. Compute nodes perform 40 iterations. In each iteration, the file access pattern

is strided with a record size of 64KBytes and a stride size of 64.

The high water mark for client-side and I/O node-side caches was varied from 0% to 100%. A high water mark value of 0% signifies that flushing is always activated, while 100% value that flushing is activated when the whole cache is full. Figures 6.35 and 6.36 show the aggregate file write throughput for 64 and 256 processes with compute phases of 0ms (left) and 500ms (right). When computing the throughput the time to close the file is included.

Note that in most cases the write throughput increases with the decrease of the high water mark on the I/O nodes. According to the intuition the best performance is obtained for 0%, i.e. when flushing is always activated. This indicates that a continuous write of dirty blocks from the I/O node to the file system is the best strategy. However, a 0% high water mark on the compute nodes does not bring performance benefits. The peaks are obtained for 6.25% and 12.5% for 64 nodes and 12.5% and 25% for 256 nodes. The values of the client-side water mark for 256 nodes are smaller than for 64 nodes, as the size of both client-side caches and I/O caches are scaled up by a factor of four, thereby reducing the data pressure in the write pipeline.

For compute phases of 500ms there is more potential for overlap between computation and I/O. However, when comparing with a 0ms compute phase, there is a significant performance increase only for small high water marks of the client-side caches. This is explained by the fact that small values of high water marks increase the probability of continuous flushing and, therefore, of overlapping I/O with computation. The efficiency of a flushing strategy can be estimated by the time to completely flush the file data at file close: the smaller the close time, the more efficient the strategy. Figure 6.37 plots in parallel the aggregate write throughput and file close time for 64 nodes, 0s and 500ms compute phase, 12.5% and 0% high water mark for the client-side cache and I/O node-side cache, respectively. The figure confirms that the aggregate throughput is inversely proportional to the close time.

File read performance

The goal of this experiment is to evaluate the performance of prefetching into the both client-side and I/O node-side caches.

The experiment was run on 64 compute nodes inside a pset. The size of client-side cache on a compute node was 128MBytes and the size of I/O node-side cache 512MBytes. All the compute nodes cache data (act as aggregators). The file block size is 4MBytes. SimParIO was configured to read a total of 10GBytes, i.e. each process repeatedly reads a record of 4MBytes in 40 phases. No views were used, a further evaluation of the prefetching based on views is presented in the next section for BTIO benchmark.

We evaluate various combinations of prefetching policies for different numbers of prefetched file blocks. For client-side caches the number of prefetched blocks were 0 (no prefetching), 4, 8, 16, and 32. On the I/O nodes this number was varied from 0 (no caching) to 128. Figure 6.35 shows the aggregate file read for compute phases of 0ms (left) and 500ms (right).

As expected, the client-side prefetching has a stronger influence on the read performance than I/O node-side prefetching. When no client-side prefetching is used, the I/O node prefetching does not appear to bring any performance benefit. Client-side prefetching brings more than one order of magnitude improvement, especially when the compute phase to be overlapped is increased to 500ms.

In order to better understand the impact on prefetching we plot in Figure 6.39 the aggregate

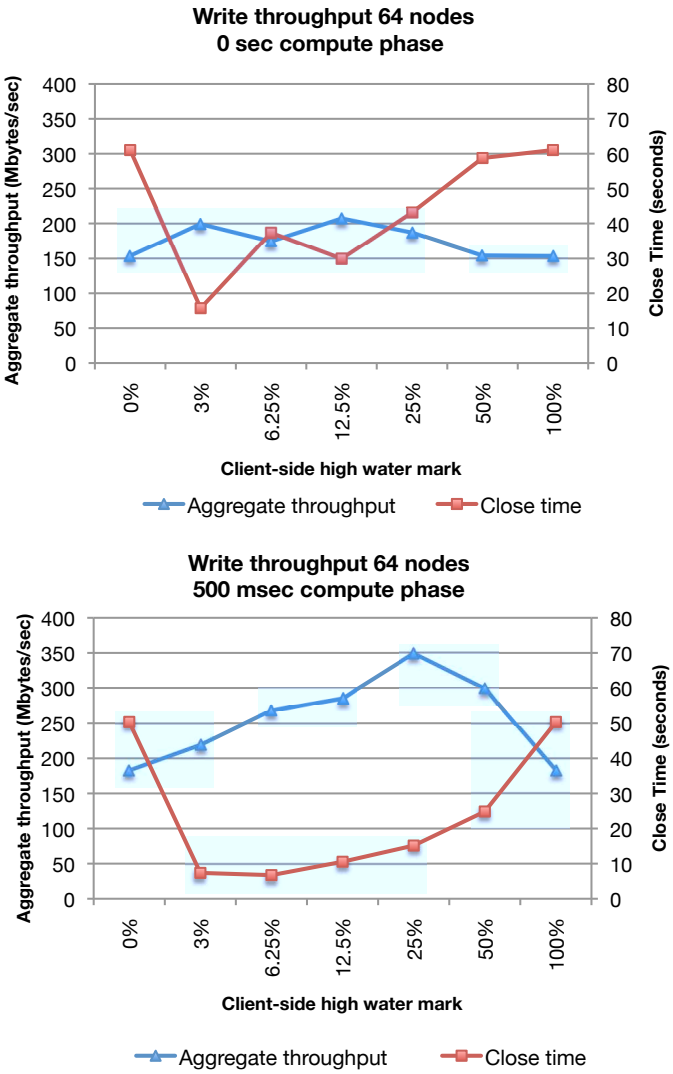

Figure 6.37: *BG/P. The figure plots in parallel the aggregate write throughput and file close time for 64 nodes, 0ms and 500ms compute phase, 12.5% and 0% high water mark for the client-side cache and I/O node-side cache, respectively. The close time can be seen as an efficiency metric of a flushing strategy: the graph shows that aggregate throughput is inversely proportional to the close time.*

read throughput of the individual 40 read phases for two cases from Figure 6.38 for 16 prefetched blocks on the I/O node for both 0ms and 500ms compute phase. We note that prefetching starts to pay off in phase 24 for no compute phase and in phase 20 for a 500ms compute phase. The prefetching is substantially more efficient when is overlapped with computation, all the phases after phase 20 appear to be served from the client-side cache.

BTIO benchmark

The goal of this section is to present an evaluation of our data staging approach for the BTIO benchmark. The client-side cache on each compute node has 64MBytes, while the I/O node-side cache has 512MBytes. All application nodes acted as aggregators. We report the results for BTIO

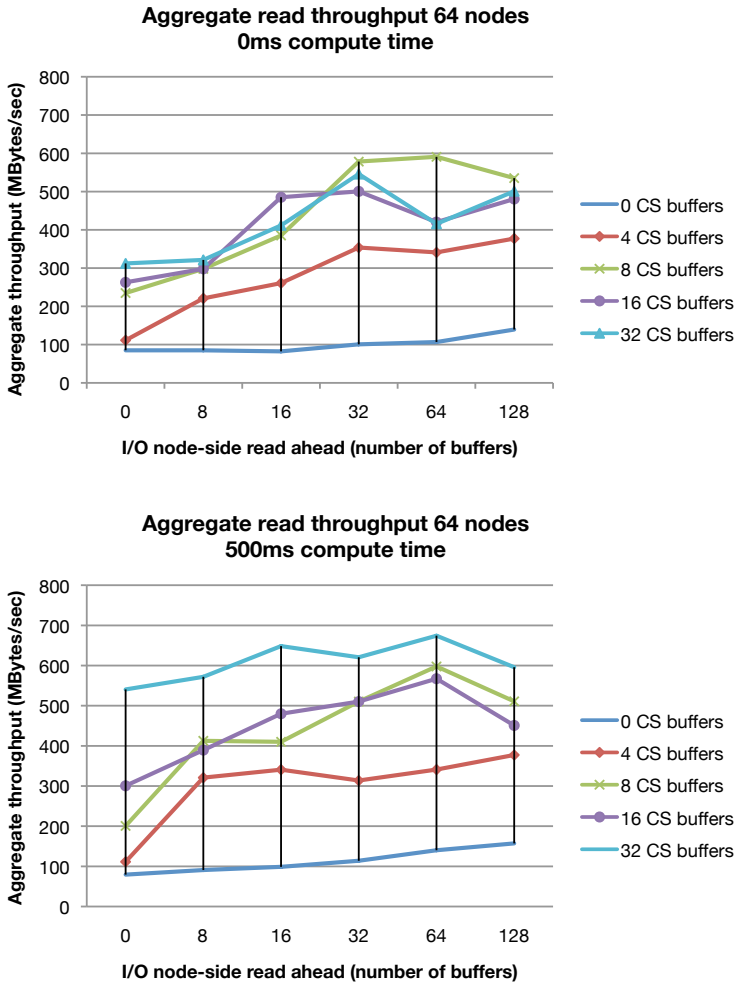

Figure 6.38: *BG/P. Effect of prefetching window on the aggregate file read throughput for a pset of 64 processes, for 0 , 4, 8, 16, and 32 prefetched client-side caches blocks and 0, 8, 16, 32, 64, and 128 prefetched I/O node-side caches blocks. Client-side prefetching has a stronger influence on the read performance than I/O node-side prefetching. It brings up to one order of magnitude improvement in presence of computation.*

class B producing a file of 1.6GBytes.

File writes. We evaluate the file writes of the BTIO benchmark for four different setups: two-phase I/O over IBM solution (CIOD), view-based I/O with no caching (VBIO), view-based I/O with client-side caching (VBIO-CS), and view-based I/O with both client-side and I/O node-side caching (VBIO-CS-IONS). Figure 6.40 shows the total time breakdown into compute time, file write time, and close time. The close time is relevant because all data is flushed to the file system when the file is closed. We notice that in all solutions the compute time is roughly the same. VBIO reduces the file write time without any asynchronous transfers. VBIO-CS reduces both the write time and close time, as data is asynchronously written from compute node to I/O node. For VBIO-CS-IONS, the network and I/O activity are almost entirely overlapped with computation. We conclude that the performance of the file writes gradually improves with the

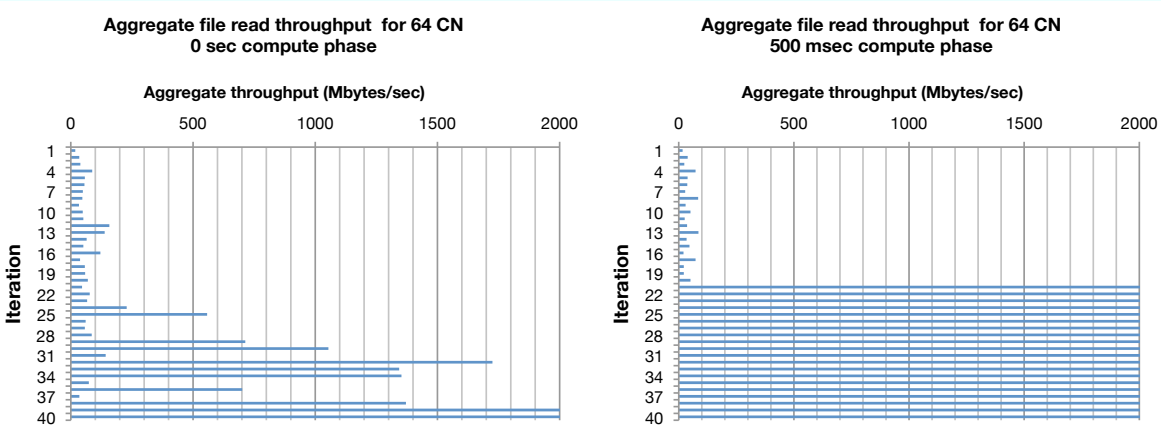

Figure 6.39: *BG/P. Histogram of the 40 phases of file read for the 16 blocks read-ahead, for 0ms and 500ms compute phase. Prefetching starts to pay off in phase 24 for no compute phase and in phase 20 for a 500ms compute phase*

increasing degree of asynchrony in the system.

File reads. BTIO performs all forty read phases in sequence, without any interleaving compute phases. In order to evaluate the effect on prefetching in presence of computation, a computation phase was inserted between consecutive read phases.

Figure 6.41 displays the file read performance without prefetching (for two-phase I/O and view based I/O) and with prefetching for 0ms, 500ms and 1000ms compute phases. We note that prefetching pays off when the client-side prefetching pool has at least 8 blocks and computation is present. The worst time was obtained for 2 prefetched file blocks and no computation and the best for 16 prefetched blocks and 1 second compute phase. Figure 6.42 shows the measured times of the 40 file read operations for 64 processes for the these two cases. We note that, in the worst case depicted on the left, the phase time decreases starting with phase 19, and in the best case shown on the right with phase 11. This indicates the timing when the read accesses start to hit the cache. In the best case, the presence of computation causes a more uniform distribution and a reduction of access times in the initial phases.

Discussion

This section targets to answer the questions from the introduction in the light of the measurements presented in the previous section. Additionally, it discusses the generality of our solution with regard to scalable systems other than Blue Gene.

The results demonstrate that a significant performance improvement can be obtained from multiple level data staging. The employment of client-side and I/O-node caches helps overlap the latency for both file writes and reads and may contribute to up to a five-hold increase for writes and an order of magnitude for reads depending on various parameters. As expected, the client-side cache contribution to the performance improvement is predominant. The applications access the client-side cache over the torus network, contributing to decrease the number of small transfer over the tree network and to better distribute the transfers over time. Therefore, this approach targets to reduce the congestion over the shared tree network.

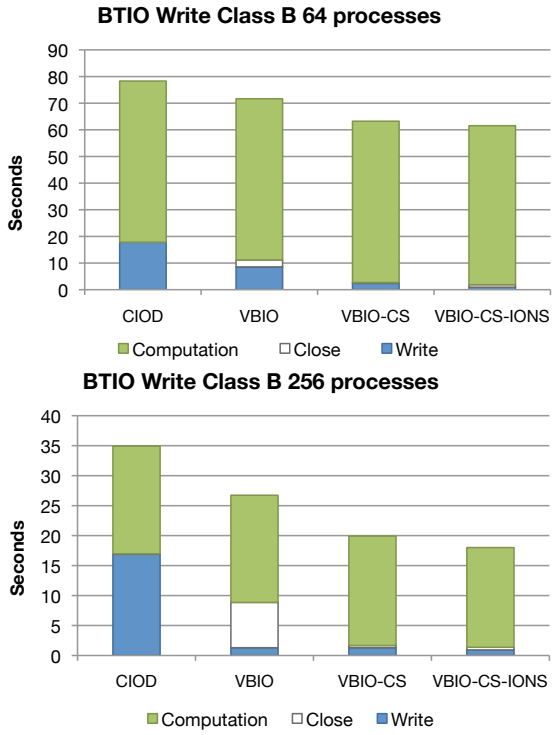

Figure 6.40: *BG/P. BTIO class B file write times for 64 and 256 processes. The performance of the file writes gradually improves with the increasing degree of asynchrony in the system: the best overlap is achieved when both client-side cache and I/O node-side cache are used.*

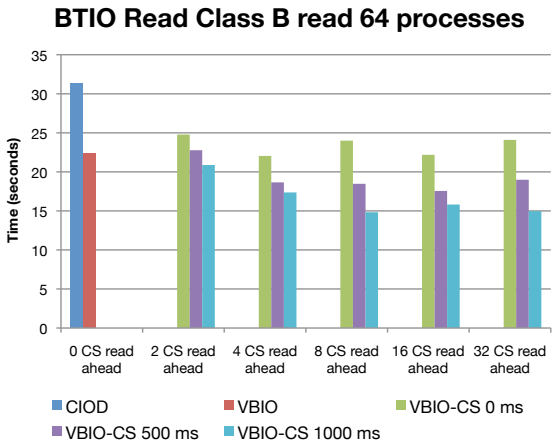

Figure 6.41: *BG/P. BTIO class B file read times for 64 processes for client-side prefetching pool sizes of 0, 2, 4, 8, 16 file blocks. Prefetching is improved only in the presence of computation and for client-side prefetching pools of at least 8 blocks.*

The write-back performance shows a strong dependence on the flushing high water mark of both client-side and I/O node-side cache. The performance difference between best and worst figures for these two parameters can be as high as two-fold. In the considered cases the best policy for I/O node appears to a combination of continuous flushing on the I/O nodes (0% high water mark), and a burst flushing on the compute nodes (high water mark greater than 0). The

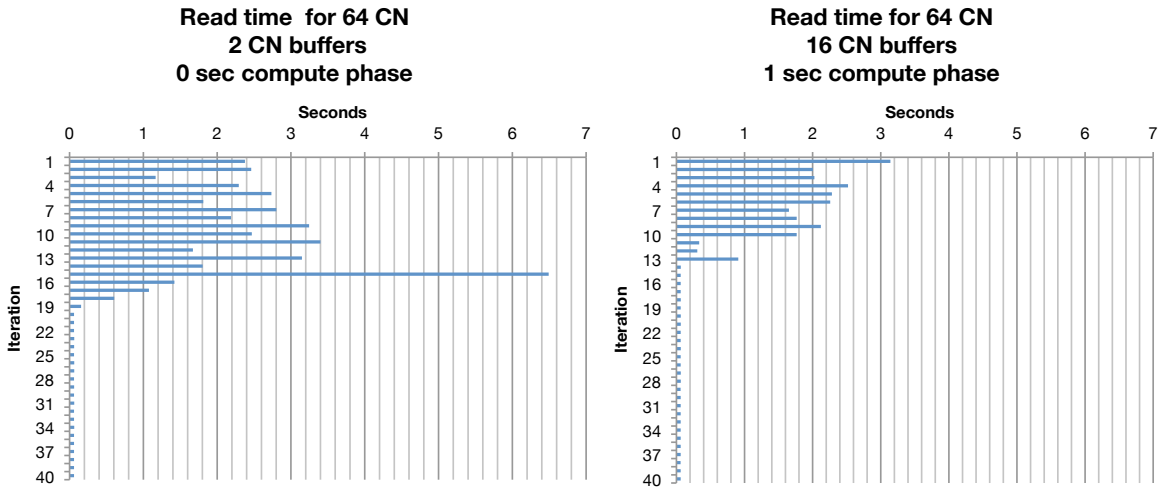

Figure 6.42: *BG/P. Histograms of the 40 file read operations of BTIO class B times for 64 processes for the best and worse performing cases from Figure 6.41. The reads start to hit the cache in phase 19 and 11, respectively.*

size of the client-side cache may also cause a performance difference as high as two-fold. However, the size of the I/O node-side cache seems to have a weak effect on performance.

The prefetching brings performance benefits of up to one order of magnitude depending on the prefetching pool sizes on both compute nodes and I/O nodes. The prefetching into the client-side cache is critical for performance in all cases. The prefetching into the I/O node is important when application read operations are not interleaved with computation. In this case the I/O node prefetching works in parallel with on-demand prefetching on the compute node, increasing the pipeline parallelism.

The client-side cache scales with the number of aggregators. The results suggest that, for efficient pipelining, client-side and I/O node-side caches have to be sized in such a way that the I/O node cache size should be at least equal to the sum of the client-side caches in the corresponding pset.

6.4 Summary

This chapter have presented an experimental evaluation of our prototypes on both clusters and supercomputers. Performance evaluation shows that the combination of collective strategies with overlapping of computation, communication, and I/O may bring a substantial performance benefit for access patterns common for parallel scientific applications.

For clusters, our experimental results show that view-based I/O can significantly reduce the total run time of a data intensive parallel application, by reducing both I/O cost and implicit synchronization cost, while requiring no extra communication time.

Additionally, the experiments demonstrate that AHPIOS offers a substantial performance benefit over the traditional MPI-IO solutions on both PVFS and Lustre parallel file systems. We notice that the AHPIOS aggregate throughput of both write and read operations scales smoothly with the number of AHPIOS servers, as in the previous case, independently if the AHPIOS servers share or not the compute node with the MPI application.

The benchmarks show that our GPFS-based solution for clusters bring satisfactory results for large files. Unlike indicated in [PTH+01], we showed that certain performance enhancement can be obtained for both write and read operations when GPFS data-shipping mode is enabled.

For Blue Gene systems, our generic cache solution reduces both the write time and close time, as data is asynchronously written from compute node to I/O node. In a complete cache solution the network and I/O activity are almost entirely overlapped with computation. We conclude that the performance of the file writes gradually improved with the increasing degree of asynchrony in the system.

Chapter 7

Final remarks and conclusions

In this thesis we have proposed a generic multi-tier cache architecture and an asynchronous data staging strategy that hides the latency of data transfer between cache tiers. We have shown that hiding file system latency may provide parallel applications a significant performance benefit and scalability.

The thesis has properly fulfilled all the primary objectives indicated in Section 1.2. Our generic parallel I/O architecture has accomplished the thesis objectives in the following ways:

Cross platform software factorization. The proposed architecture provides a parallel I/O middleware that allows integrating optimizations at different levels. The architecture allows combining different scenarios, depending of the I/O system. In Section 4.2, we have presented AHPIOS for file-system independent parallel I/O architectures. In Section 4.3, we have shown how popular file systems, like GPFS, can take advantage from our generic parallel I/O architecture.

High-performance parallel I/O. Our experimental evaluation demonstrates that our proposed architecture offers a substantial performance benefit over the traditional MPI-IO solutions on both clusters and supercomputers. Performance evaluation has shown in sections 6.2.2, 6.2.3, and 6.3.2, that the combination of collective strategies with overlapping of computation, communication, and I/O may bring a substantial performance benefit for access patterns common for parallel scientific applications such as BTIO and Flash.

Scalability. Our parallel I/O architecture targets to scalability for both clusters and supercomputers. First, in Section 6.2.2 we noticed that the aggregated, throughput of both write and read operations scales smoothly with the number of AHPIOS servers on clusters, as we proof in Section 6.2.2. Second, as shown in Section 6.3.2, we have demonstrated that our solution for supercomputers outperforms the native approaches in all of the cases, especially when applications increase the number of compute nodes and aggregators.

Dynamic virtualization. We have proposed view-based I/O as a file-independent optimization for parallel file systems. Additionally, in Section 4.2 we have described the design of AHPIOS, which offers storage virtualization on-the-fly. Different parallel file systems can be virtualized in order to increase the parallelism degree. For example in Chapter 6 we have demonstrated how

AHPIOS can virtualize the local storage system in a parallel I/O system.

Portability. In order to guarantee portability, our client-side solutions (view-based I/O, GPFS-based I/O, and AHPIOS) are completely implemented in MPI, allowing on any cluster system. Additionally, AHPIOS is a suitable solution not only for clusters of computers, but also for supercomputers, grids, and clouds. In order to deploy our optimizations in different supercomputer, we have used ZeptoOS, which allows a competitive deployment of ZOIDFS in massively parallel architectures not only in Blue Gene, but also in Cray.

The rest of this chapter is organized as follows. We start by describing the contributions of this thesis. Then, we enumerate the publications obtained, and finally we propose new lines of research arising from this thesis.

7.1 Contributions

This thesis makes the following contributions:

Generic parallel I/O architecture. This thesis presents a scalable parallel I/O architecture targeting to transparently hide the latency of file system accesses to the applications in both cluster and supercomputer systems. Given the increasing hierarchy of networks involved in file accesses, our solution is designed to maximize the degree of overlapping between computation, file I/O-related communication, and file system access.

File system-independent collective I/O method. This thesis presents and evaluates view-based I/O, a novel file system-independent I/O optimization of parallel file accesses. View-based I/O differs from existing methods in that it optimizes the I/O transfers in three ways. First, file access patterns extracted from process file views are leveraged in order to reduce the number of file metadata operations. Second, a structured I/O reduces the size of metadata transfers. Third, the data locality is increased through a distributed file cache stored in RAM on the compute nodes.

Ad-hoc virtualized parallel I/O system. This thesis presents and evaluates AHPIOS, the first scalable parallel I/O system completely implemented in Message Passing Interface (MPI) in a portable manner. AHPIOS allows parallel applications to dynamically manage and scale distributed partitions in a convenient way, eliminating the need for the traditional parallel file systems. The configurations of client I/O library and storage management system are unified and allow for a tight integration of the optimizations of these layers. AHPIOS can be used as a light-weight low-cost alternative to any parallel file system, virtualizing on-demand independent storage resources.

Novel multi-layer techniques for latency hiding of parallel I/O. This thesis introduces and evaluates novel multi-layer data staging techniques for both clusters and supercomputers. We describe and evaluate a two-level hierarchy for clusters (in Chapter 4) and Blue gene system (in Chapter 5) consisting of client-side and I/O node-side caching. The file cache management modules coordinate the data staging between application and storage through the Blue Gene networks. The experimental results (sections 6.2 and 6.3) demonstrate that our architecture achieves significant performance improvements through a high degree of overlapping between computation, communication, and file I/O.

7.2 Thesis results

The principal contributions of the thesis have been published in diverse papers in international conferences and journals. We enumerate the publications classified in five groups: articles in journals, book chapters, posters, international, and national conferences.

- Journals
  - *Implementation and Evaluation of File Write-Back and Prefetching for MPI-IO over GPFS*. Javier García Blas, Florin Isaila, Jesús Carretero, David Singh, and Felix García-Carballeira. Special issue in International Journal of High Performance Computing. 2010 Impact Factor: 1.824
  - *A scalable MPI implementation of an ad-hoc parallel I/O system*. Florin Isaila, Javier García Blas, Jesús Carretero, Wei-keng Liao and Alok Choudhary. International Journal of High Performance Computing. 2008 Impact Factor: 1.824
- International conferences
  - *Multiple-level MPI file write-back and prefetching for Blue Gene systems*. Javier García Blas, Florin Isaila, Jesús Carretero, Robert Latham and Robert Ross. 16th EuroPVM/MPI, Finland, September, 2009.
  - *A General Parallel I/O Architecture for Clusters and Supercomputers*. Javier García Blas, Florin Isaila and Jesús Carretero. IEEE International Parallel and Distributed Processing Symposium (IPDPS), TCPP PhD Forum, Rome, Italy, May, 2009.
  - *Latency hiding file I/O for Blue Gene systems*. Florin Isaila, Javier García Blas, Jesús Carretero, Rob Latham, Sam Lang, Rob Ross. 9th IEEE International Symposium on Cluster Computing and the Grid (CCGRID), Shanghai, May, 2009.
  - *A scalable view-based collective I/O optimization for large-scale parallel applications*. Javier García Blas, Florin Isaila and Jesús Carretero. TAM workshop. Stuttgart, Germany, December, 2008.
  - *AHPIOS: An MPI-based ad-hoc parallel I/O system*. Florin Isaila, Javier García Blas, Jesús Carretero, Wei-keng Liao, Alok Choudhary. 14th Intl Conference on Parallel and Distributed Systems, Melbourne, Australia, September, 2008.
  - *Implementation and evaluation of an MPI-IO interface for GPFS in ROMIO*. Javier García Blas, Florin Isaila, Jesús Carretero and Thomas Grossemann. The 15th Euro PVM/MPI 2008 conference, Dublin, Ireland, September, 2008.
  - View-based collective I/O for MPI-IO. Javier García Blas, Florin Isaila, David E. Singh and Jesús Carretero. IEEE International Symposium on Cluster Computing and the Grid (CCGRID), Lyon, France, 2008.
  - *WiP-FAST08 View-based collective I/O for MPI-IO*. Javier García Blas, Florin Isaila, Jesús Carretero. 6th USENIX Conference on File and Storage Technologies (FAST '08), San Jose, California, USA, 2008.

- Book chapters
  - *A general parallel I/O architecture for massively parallel supercomputers.* Javier García Blas, Florin Isaila, Jesús Carretero, ACACES 2009, Terrasa, Spain, July, 2009, Academia Press, 978 90 382 14, 277-280
  - *A view-based approach for collective I/O operations.* Javier García Blas, Florin Isaila, Jesús Carretero, Transnational Access Meeting 2008, Bologna, Italy, June, 2008
- Posters
  - *A general parallel I/O architecture for clusters and supercomputers.* Javier García Blas, Florin Isaila, Jesús Carretero. IEEE International Parallel and Distributed Processing Symposium (IPDPS), TCPP PhD Forum, Rome, Italy, May, 2009.
  - *View-based collective I/O for MPI-IO.* Javier García Blas, Florin Isaila y Jesús Carretero. 6th USENIX Conference on File and Storage Technologies (FAST '08), San Jose, California, USA, 2008.
- National conferences
  - *Arquitectura de E/S paralela de alta prestaciones para sistemas Blue Gene.* Javier García Blas, Florin Isaila y Jesús Carretero. XX Jornadas de Paralelismo, A Coruna, Spain, September, 2009.
  - *Implementación y evaluación de una interfaz para GPFS en ROMIO.* Javier García Blas, Florin Isaila, Jesús Carretero. Jornadas de paralelismo de Castellón, Castellón, Spain, September, 2008.
  - *Operaciones colectivas basadas en vistas para sistemas de ficheros paralelos.* Javier García Blas, F. Isaila, Jesús Carretero. Jornadas de paralelismo de Zaragoza, Zaragoza, Spain, September, 2007

Other achievements in this thesis include research stays, seminars, and research grants:

- Research stays
  - Mathematics and Computer Science Division (MCS) at Argonne National Laboratory. Hosted by Robert Lathan and Robert Ross. Fall 2008 Chicago (USA). Duration: 3 months.
  - High Performance Computing Center Stuttgart (HLRS). Hosted by Alexander Schulz and Rainer Keller. Fall 2007, Stuttgart (Germany). Duration: 3 months.
- Seminars
  - A Scalable View-Based Collective I/O Optimization for Large-Scale Parallel Applications. Mathematics and Computer Science Division (MCS) at Argonne National Laboratory (ANL). Chicago (USA), October 10, 2008.

- Research grants
  - TAM workshop. Stuttgart, Germany, December 2008. Funds: accomodation and travel cost.
  - Programa propio de investigación, ayudas de movilidad. University Carlos III. Funds: 2,135€
  - Supercomputing 08. Texas, UE, November 2008. Student Volunteer Program. Funds: accommodation cost and conference registration.
  - Conference on File and Storage Technologies (FAST '08). San Jose, California, USA. USENIX Association student grants. Funds: 1,101€ and registration.
  - High Performance Computing Center Stuttgart (HLRS). Stuttgart, Germany, December 2007. Funds: 1,500€ + travel cost.

7.3 Future directions

There are several lines of research arising from this work which could be pursued.

7.3.1 Supercomputers

The architecture of large-scale supercomputers such as IBM Blue Gene/L, IBM Blue Gene/P, and Cray XT systems is organized by specializing the system into disjoint sets of compute and I/O nodes. The compute nodes are assigned to an application through a batch scheduler. Dedicated I/O nodes serve each set of compute nodes. Therefore, the I/O nodes corresponding to the assigned compute nodes are known after the job is scheduled. AHPIOS servers can be spawned after the scheduling is performed. Here we describe our initial experiences of AHPIOS on Blue Gene systems.

Blue Gene systems currently do not offer MPI dynamic process management. In our initial setup, I/O daemons are started on the I/O nodes as MPI processes. Each AHPIOS server is a thread of an I/O daemon running on the I/O node. The MPI-IO calls are forwarded through the tree network to the AHPIOS servers, where they are processed in cooperation. The communication among AHPIOS servers is performed through MPI calls and goes over the switched network interconnecting the I/O nodes. We have managed to deploy the AHPIOS on the Blue Gene/L system in Argonne National Laboratory (ANL), and we are currently experimenting with the new Blue Gene/P system installed there.

7.3.2 Parallel I/O architecture for multi-core systems

In the current Top 500 supercomputer list, majority of processors belong to the multi-core processor family; 76.6% of the listed systems are quad-core processors. For example, there are commercial products that have 8 cores on the same chip, and there is a research prototype with 80 cores [VHR+08]. Furthermore, the next-generation of Blue Gene supercomputers (BG/Q) will have 8 or even 16 cores per node, with 1 GByte of memory per core.

The popularity of multi-core processors provides a flexible solution to increase the computational capability of clusters and supercomputers. Parallel applications can benefit from multi-core

processors [Gee05]. Although the system performance may improve with multi-core processor, I/O requests initiated by multiple cores may saturate the I/O systems, and furthermore increase the latency of file accesses in parallel applications [LGBK08, CBS+08a].

We propose to extend our generic parallel I/O architecture in order to support massively multi-core architectures. It will be necessary to include a new file cache tier, even more closed to applications, in order to aggregate file accesses inside nodes.

7.3.3 Cloud computing

Many researchers agree that in the future companies will rely less on their own infrastructures and more on remote clouds. A cloud is defined as Infrastructure as a Service (IaaS). Amazon [htt08a] provides pay-per-use computing and storage resources through Web Services. With Amazon Elastic Computing Cloud (EC2) the users may run instances of virtual machines consisting of CPU, memory, and local disks. The cloud is elastic: the users can increase or decrease their computation needs by acquiring or releasing VM instances. The local disks store the information only during the life of an instance. For permanent storage, the users should use either the Elastic Block Store (EBS) or Simple Storage System (S3). EBS is an elastic (could be increased or decreased) storage partition on top of which a file system could be installed. EBS can be mounted by several instances. S3 is a persistent storage service, on which users can store objects. Storing data on the local storage of each instance comes at no extra cost, while there is a charge for storing both on EBS and S3. Other academic or research cloud projects such as Nimbus [htt08b] and Stratus [htt08c] offer EC2-like functionality. However, none of these projects offers a parallel I/O distributed system such as AHPIOS.

AHPIOS could be used in two ways with EC2-like services. First, AHPIOS can offer an MPI-IO integrated shared distributed partition based on the local storage of several available instances. This provides an efficient on-demand parallel I/O system to MPI applications running on the available instances. Additionally, AHPIOS can be simply scaled up or down by a simple restart. However, in this case the data from the local storage, has to be backed up on either EBS or S3, for instance through an asynchronous data staging running in background. Second, AHPIOS could be used to efficiently store data in parallel over several EBS partitions. Currently, we are investigating both possibilities and we plan to implement and evaluate both of them in a near future.

7.3.4 High-performance POSIX interface

The POSIX I/O API is the most widely used API for access to file systems, both by applications as well as by I/O libraries. Many parallel applications commonly access non-contiguous data regions in the file through POSIX interface. Accessing files in a non-contiguous fashion involves lots of POSIX accesses.

Our caching methods could be used to optimize non-contiguous accesses by aggregating small contiguous pieces of data into client-side file cache. We can integrate our techniques into FUSE [htt09b], which offers an user-level POSIX interface. This approach will allow using FUSE-based POSIX interface instead of MPI-IO

7.3.5 Parallel I/O system for Windows platforms

According to Top 500 list, there is a growing number of cluster and supercomputers that use Windows platforms. GPFS and Lustre can be used as a parallel I/O solution for these environments. However these parallel file systems may incur large overheads for deploying, configuring, and achieving scalability.

There is a limited amount of research on high-performance parallel I/O for Windows. WinPFS [Jos04] is a parallel file system integrated within the Windows kernel. WinPFS leverages existing standard client and server components on the Windows platform (ie. NFS [LD02] and CIFS [Her03]), in order to offer transparent parallel access to files distributed over multiple remote disks. The disadvantage is that the user can not control the striping size of a file across nodes.

Since MPI distributions like MPICH2 can be deployed on Windows platforms, we propose to use AHPIOS as a Windows parallel I/O system. AHPIOS can offer shared distributed partitions based on the local storage (local NTFS) or CIFS. This provides an efficient on-demand parallel I/O system to Windows-based MPI applications running on the available instances. Additionally, AHPIOS can be simply scaled up or down by a simple restart. Finally, optimizations proposed in this thesis like view-based I/O can be used as well.

Bibliography

[Aba03] J. H. Abawajy. Performance Analysis of Parallel I/O Scheduling Approaches on Cluster Computing Systems. In *CCGRID '03: Proceedings of the 3st International Symposium on Cluster Computing and the Grid*, page 724, Washington, DC, USA, 2003. IEEE Computer Society.

[ABB+08] S. Alam, R. Barrett, M. Bast, M. R. Fahey, J. Kuehn, C. McCurdy, J. Rogers, P. Roth, R. Sankaran, J. S. Vetter, P. Worley, and W. Yu. Early evaluation of IBM BlueGene/P. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–12, Piscataway, NJ, USA, 2008. IEEE Press.

[ACI+09] Nawab Ali, Philip Carns, Kamil Iskra, Dries Kimpe, Samuel Lang, Robert Latham, Robert Ross, Lee Ward, and P. Sadayappan. Scalable I/O Forwarding Framework for High-Performance Computing Systems. In *Proceedings of IEEE Conference on Cluster Computing, New Orleans, LA*, September 2009.

[AHP01] Nicholas K. Allsopp, John F. Hague, and Jean-Pierre Prost. Experiences in Using MPI-IO on Top of GPFS for the IFS Weather Forecast Code. In *Euro-Par '01: Proceedings of the 7th International Euro-Par Conference Manchester on Parallel Processing*, pages 380–384, London, UK, 2001. Springer-Verlag.

[AKB+07] Sadaf R. Alam, Jeffery A. Kuehn, Richard F. Barrett, Jeff M. Larkin, Mark R. Fahey, Ramanan Sankaran, and Patrick H. Worley. Cray XT4: an early evaluation for petascale scientific simulation. In *SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, pages 1–12, New York, NY, USA, 2007. ACM.

[AR] GPFS Administration and Programming Reference. *http://publib.boulder.ibm.com/infocenter/.*

[Ass98] InfiniBand Trade Association. *Infiniband architecture specification*, 1998.

[BCS+08] Surendra Byna, Yong Chen, Xian-He Sun, Rajeev Thakur, and William Gropp. Parallel I/O prefetching using MPI file caching and I/O signatures. In *SC '08*, pages 1–12, 2008.

[BMV03] R. Badrinath, C. Morin, and G. Valle. Checkpointing and recovery of shared memory parallel applications in a cluster. In *Proc. Intl. Workshop on Distributed Shared Memory on Clusters (DSM 2003)*, 2003.

[Bor97] Rajesh Bordawekar. Implementation of Collective I/O in the Intel Paragon Parallel File System: Initial Experiences. In *Proc. 11th International Conference on Supercomputing*, July 1997.

[BOSS07] Julian Borrill, Leonid Oliker, John Shalf, and Hongzhang Shan. Investigation of leading HPC I/O performance using a scientific-application derived benchmark. In *SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, pages 1–12, New York, NY, USA, 2007. ACM.

[Buy99] Rajkumar Buyya. *High Performance Cluster Computing: Architectures and Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1999.

[CACR95] P.E. Crandall, R.A. Aydt, A.A. Chien, and D.A. Reed. Input/Output Characteristics of Scalable Parallel Applications. In *Proceedings of Supercomputing '95*, 1995.

[CBS+08a] Yong Chen, Surendra Byna, Xian-He Sun, Rajeev Thakur, and William Gropp. Exploring Parallel I/O Concurrency with Speculative Prefetching. In *ICPP '08: Proceedings of the 2008 37th International Conference on Parallel Processing*, pages 422–429, Washington, DC, USA, 2008. IEEE Computer Society.

[CBS+08b] Yong Chen, Surendra Byna, Xian-He Sun, Rajeev Thakur, and William Gropp. Hiding I/O latency with pre-execution prefetching for parallel applications. In *SC '08*, pages 1–10, 2008.

[CCL+02] Avery Ching, Alok Choudhary, Wei Keng Liao, Rob Ross, and William Gropp. Noncontiguous I/O through PVFS. In *CLUSTER '02*, page 405, Washington, DC, USA, 2002. IEEE Computer Society.

[CCL+03] Avery Ching, Alok Choudhary, Wei Keng Liao, Robert Ross, and William Gropp. Efficient Structured Data Access in Parallel File Systems. In *Proceedings of the IEEE International Conference on Cluster Computing*, December 2003.

[CDT06] Kai-Dee Chu, Liping Di, and Peter Thornton. Introduction of Grid Computing Application Projects at the NASA Earth Science Technology Office. In *GPC*, pages 289–298, 2006.

[CF96] P.F. Corbett and D.G. Feitelson. The Vesta Parallel File System. *ACM Transactions on Computer Systems*, 1996.

[CG99] F.W. Chang and G.A. Gibson. Automatic I/O Hint Generation Through Speculative Execution. In *Proceedings of OSDI*, 1999.

[CGL97] Toni Cortes, Sergi Girona, and Jesús Labarta. Avoiding the Cache-Coherence Problem in a Parallel/Distributed File System. In *Proceedings of High-Performance Computing and Networking*, pages 860–869, April 1997.

[CGS03] P. Cornillon, J. Gallagher, and T. Sgouros. Opendap: Accessing data in a distributed, heterogeneous environment. volume 2, pages 164–174, 2003.

[CKV93] K.M. Curewitz, P. Krishnan, and J.S. Vitter. Practical prefetching via data compression. In *Proceedings of SIGMOD Conference*, 1993.

[CONP07] Lei Chai, Xiangyong Ouyang, Ranjit Noronha, and Dhabaleswar K. Panda. pnfs/pvfs2 over infiniband: early experiences. In *PDSW '07: Proceedings of the 2nd international workshop on Petascale data storage*, pages 5–11, New York, NY, USA, 2007. ACM.

[CSM+96] J. Carretero, F.P. Serez, P. Miguel, F. Garcia, and L. Alonso. ParFiSys: A Parallel File System for MPP. *ACM SIGOPS*, 30(2), 1996.

[DHJ+04] Kei Davis, Adolfy Hoisie, Greg Johnson, Darren J. Kerbyson, Mike Lang, Scott Pakin, and Fabrizio Petrini. A performance and scalability analysis of the blue-gene/l architecture. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 41, Washington, DC, USA, 2004. IEEE Computer Society.

[DL08] Phillip Dickens and Jeremy Logan. Towards a High Performance Implementation of MPI-IO on the Lustre File System. In *OTM '08: Proceedings of the OTM 2008 Confederated International Conferences, CoopIS, DOA, GADA, IS, and ODBASE 2008. Part I on On the Move to Meaningful Internet Systems:*, pages 870–885, Berlin, Heidelberg, 2008. Springer-Verlag.

[dRBC93] J. del Rosario, R. Bordawekar, and A. Choudhary. Improved parallel I/O via a two-phase run-time access strategy. In *Proc. of IPPS Workshop on Input/Output in Parallel Computer Systems*, 1993.

[DT99] Phillip Dickens and Rajeev Thakur. Improving collective I/O performance using threads. In *In Proceedings of the 13th IPPS*, pages 38–45, 1999.

[Edg04] Edgar Gabriel and Graham E. Fagg and George Bosilca and Thara Angskun and Jack J. Dongarra and Jeffrey M. Squyres and Vishal Sahay and Prabhanjan Kambadur and Brian Barrett and Andrew Lumsdaine and Ralph H. Castain and David J. Daniel and Richard L. Graham and Timothy S. Woodall . Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[FKL+08] R. T. Fisher, L. P. Kadanoff, D. Q. Lamb, A. Dubey, T. Plewa, A. Calder, F. Cattaneo, P. Constantin, I. Foster, M. E. Papka, S. I. Abarzhi, S. M. Asida, P. M. Rich, C. C. Glendening, K. Antypas, D. J. Sheeler, L. B. Reid, B. Gallagher, and S. G. Needham. Terascale turbulence computation using the flash3 application framework on the ibm blue gene/l system. *IBM J. Res. Dev.*, 52(1/2):127–136, 2008.

[FLA08] Mark R. Fahey, Jeff M. Larkin, and Joylika Adams. I/O performance on a massively parallel Cray XT3/XT4. In *IPDPS*, pages 1–12. IEEE, 2008.

[FOR+00] B. Fryxell, K. Olson, P. Ricker, F. X. Timmes, M. Zingale, D. Q. Lamb, P. MacNeice, R. Rosner, and H. Tufo. FLASH: An Adaptive Mesh Hydrodynamics Code for Modelling Astrophysical Thermonuclear Flashes. *Astrophysical Journal Suppliment*, pages 131–273, 2000.

[GCCC+03] Félix Garcia-Carballeira, Alejandro Calderon, Jesus Carretero, Javier Fernandez, and Jose M. Perez. The Design of the Expand Parallel File System. *The International Journal of High Performance Computing Applications*, 17(1):21–38, 2003.

[Gee05] David Geer. Industry Trends: Chip Makers Turn to Multicore Processors. *Computer*, 38(5):11–13, 2005.

[GG97] Jim Gray and Goetz Graefe. The five-minute rule ten years later, and other computer storage rules of thumb. In *SIGMOD Rec.*, volume 26, pages 63–68, New York, NY, USA, 1997. ACM.

[GKL95] W. Gropp, E. Karrels, and E. Lusk. MPE graphics: scalable X11 graphics in MPI. In *Proceedings of the 1994 Scalable Parallel Libraries Conference: October 12–14, 1994, Mississippi State University, Mississippi*, pages 49–54, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1995. IEEE Computer Society Press.

[GP87] Jim Gray and Franco Putzolu. The 5 minute rule for trading memory for disc accesses and the 10 byte rule for trading memory for CPU time. In *SIGMOD '87: Proceedings of the 1987 ACM SIGMOD international conference on Management of data*, pages 395–398, New York, NY, USA, 1987. ACM.

[HDF] HDF5 home page. *http://hdf.ncsa.uiuc.edu/HDF5.*

[HER+95] J.V. Huber, C.L. Elford, D.A. Reed, A.A. Chien, and D.S. Blumenthal. PPFS: A High Performance Portable File System. In *Proceedings of the 9th ACM International Conference on Supercomputing*, 1995.

[Her03] Christopher Hertel. *Implementing CIFS: The Common Internet File System.* Prentice Hall Professional Technical Reference, 2003.

[HH05] Dean Hildebrand and Peter Honeyman. Exporting Storage Systems in a Scalable Manner with pNFS. *Mass Storage Systems and Technologies, IEEE / NASA Goddard Conference on*, 0:18–27, 2005.

[HH07] Dean Hildebrand and Peter Honeyman. Direct-pnfs: scalable, transparent, and versatile access to parallel file systems. In *HPDC '07: Proceedings of the 16th international symposium on High performance distributed computing*, pages 199–208, New York, NY, USA, 2007. ACM.

[HP] HP MPI home page. *http://www.hp.com.*

[hs95] http://www.unix systems.org/. *The Portable Operating System Interface*, 1995.

[htt08a] http://aws.amazon.com/. *Amazon web services site.*, 2008.

[htt08b] http://workspace.globus.org/clouds/nimbus.html. *Nimbus Cloud Project.*, 2008.

[htt08c] http://www.acis.ufl.edu/vws/. *Stratus Cloud Project.*, 2008.

[htt08d] http://www.panasas.com/panfs.html. *PanFS web site.*, 2008.

[htt09a] http://flash.uchicago.edu. *FLASH3 code.*, 2009.

[htt09b] http://fuse.sourceforge.net. *FUSE Homepage.*, 2009.

[hu08] http://www unix.mcs.anl.gov/zeptoos/. *ZeptoOs Project.*, 2008.

[IBM98] IBM. *GPFS: A Parallel File System*, 1998.

[Inc02] Cluster File Systems Inc. Lustre: A scalable, high-performance file system. Cluster File Systems Inc. white paper, version 1.0, November 2002. http://www.lustre.org/docs/whitepaper.pdf.

[IRYB08] Kamil Iskra, John W. Romein, Kazutomo Yoshii, and Pete Beckman. ZOID: I/O-forwarding infrastructure for petascale architectures. In *PPoPP '08*, pages 153–162, 2008.

[ISC+06] Florin Isaila, David Singh, Jesús Carretero, Félix Garcia, Gábor Szeder, and Thomas Moschny. Integrating logical and physical file models in the MPI-IO implementation for Clusterfile. In *Sixth IEEE International Symposium on Cluster Computing and the Grid (CCGRID)*. IEEE Computer Society, 2006.

[ISCG06] Florin Isaila, David Singh, Jesús Carretero, and Félix Garcia. On evaluating decentralized parallel I/O scheduling strategies for parallel file systems. In *VECPAR 2006*, 2006.

[IT01] F. Isaila and W. Tichy. Clusterfile: A flexible physical layout parallel file system. In *First IEEE International Conference on Cluster Computing*, October 2001.

[IT03a] F. Isaila and W. Tichy. Clusterfile: A flexible physical layout parallel file system. *Concurrency and Computation: Practice and Experience*, 15(7–8):653–679, 2003.

[IT03b] F. Isaila and W. Tichy. View I/O:improving the performance of non-contiguous I/O. In *Third IEEE International Conference on Cluster Computing*, pages 336–343, December 2003.

[J. 09] J. H. Chen, A. Choudhary, B. de Supinski, M. DeVries, E. R. Hawkes, S. Klasky , W. K. Liao, K-L. Ma, J. Mellor-Crummey, N. Podhorszki, R. Sankaran and S. Shende, and C. S. Yoo. Terascale direct numerical simulations of turbulent combustion using s3d. volume 2, January-March 2009.

[Jos04] Jose Maria Perez, Jesus Carretero, and Jose Daniel Garcia. A Parallel File System for Networks of Windows Worstations. volume 0, 2004.

[JSWB97] Ravi Jain, Kiran Somalwar, John Werth, and J. C. Browne. Heuristics for scheduling I/O operations. *IEEE Transactions on Parallel and Distributed Systems*, 8(3):310–320, March 1997.

[KE93] D. Kotz and C.S. Ellis. Practical prefetching techniques for multiprocessor file systems. *Journal of Distributed and Parallel Databases*, 1993.

[KMM94] Gene H. Kim, Ronald G. Minnich, and Larry Mcvoy. Bigfoot-NFS: A Parallel File-Striping NFS Server (Extended Abstract), 1994.

[Kot94] D. Kotz. Disk-directed I/O for MIMD Multiprocessors. In *Proc. of the First USENIX Symp. on Operating Systems Design and Implementation*, 1994.

[KTP+96] T. Kimbrel, A. Tomkins, R.H. Patterson, B. Bershad, P. Cao, E.W. Felten, G.A. Gibson, A.R. Karlin, and K. Li. Trace-Driven Comparison of Algorithms for Parallel Prefetching and Caching. In *Proc. of the 2nd USENIX Symposium on OSDI*, 1996.

[KV02] M. Kallahalla and P.J. Varman. PC-OPT: optimal offline prefetching and caching for parallel I/O systems. *Computers, IEEE Transactions on*, 51(11):1333–1344, Nov 2002.

[LCC+05] Wei Keng Liao, Kenin Coloma, Alok Choudhary, Lee Ward, Eric Russel, and Sonja Tideman. Collective Caching: Application-Aware Client-Side File Caching. In *Proceedings of the 14th International Symposium on High Performance Distributed Computing (HPDC)*, July 2005.

[LD02] P. Lombard and Y. Denneulin. NFSP: a distributed NFS server for clusters of workstations. In *Parallel and Distributed Processing Symposium., Proceedings International, IPDPS 2002, Abstracts and CD-ROM*, pages 35–40, 2002.

[LGBK08] Guangdeng Liao, Danhua Guo, Laxmi Bhuyan, and Steve R King. Software techniques to improve virtualized i/o performance on multi-core systems. In *ANCS '08: Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, pages 161–170, New York, NY, USA, 2008. ACM.

[LLC+03] Jianwei Li, Wei Keng Liao, Alok Choudhary, Robert Ross, Rajeev Thakur, William Gropp, Rob Latham, Andrew Siegel, Brad Gallagher, and Michael Zingale. Parallel netCDF: A High-Performance Scientific I/O Interface. In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 39, Washington, DC, USA, 2003. IEEE Computer Society.

[Lon] Lonestar home page. *http://www.tacc.utexas.edu/services/-userguides/lonestar/*.

[LR99] W.B. Ligon and R.B. Ross. An Overview of the Parallel Virtual File System. In *Proceedings of the Extreme Linux Workshop*, June 1999.

[MCFX97] Sachin More, Alok N. Choudhary, Ian T. Foster, and Ming Q. Xu. MTIO - A Multi-Threaded Parallel I/O System. In *IPPS '97: Proceedings of the 11th International Symposium on Parallel Processing*, pages 368–373, Washington, DC, USA, 1997. IEEE Computer Society.

[Mea06] José Moreira and et al. Designing a highly-scalable operating system: the Blue Gene/L story. In *SC '06*, page 118, 2006.

[MEFT96] Z. B. Miled, R. Eigenmann, J. A. B. Fortes, and V. Taylor. Hierarchical processors-and-memory architecture for high performance computing. In *FRONTIERS '96: Proceedings of the 6th Symposium on the Frontiers of Massively Parallel Computation*, page 355, Washington, DC, USA, 1996. IEEE Computer Society.

[Mes95] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*, 1995.

[Mes97] Message Passing Interface Forum. *MPI2: Extensions to the Message Passing Interface*, 1997.

[MPI] MPI tile I/O. *http://www-unix.mcs.anl.gov/pio-benchmark/*.

[MPI95] MPI Forum, http://www-unix.mcs.anl.gov/mpi/mpich/. *MPICH website*, 1995.

[MWLY03] Xiaosong Ma, Marianne Winslett, Jonghyun Lee, and Shengke Yu. Improving MPI-IO Output Performance with Active Buffering Plus Threads. In *IPDPS*, pages 22–26, 2003.

[Myr] Myricom. GM: the low-level message-passing system for Myrinet networks. *http://www.myri.com/*.

[NEC] NEC MPI home page. *http://www.nec.com*.

[NK97] N. Nieuwejaar and D. Kotz. The Galley Parallel File System. *Parallel Computing*, pages 447–476, 1997.

[NKP+96] N. Nieuwejaar, D. Kotz, A. Purakayastha, C.S. Ellis, and M.L. Best. File Access Characteristics of Parallel Scientific Workloads. In *IEEE Transactions on Parallel and Distributed Systems, 7(10)*, pages 1075–1089, October 1996.

[OT04] V. Olaru and W.F. Tichy. On the Design and Performance of Remote Disk Drivers for Clusters of PCs. In *Proceedings of PDPTA'04 - The 2004 International Conference on Parallel and Distributed Processing Techniques and Applications*, June 2004.

[PCG+97] F. Pérez, J. Carretero, F. García, P. De Miguel, and L. Alonso. Evaluating parfisys: a high-performance parallel and distributed file system. *J. Syst. Archit.*, 43(8):533–542, 1997.

[PGG+95] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed prefetching and caching. *SIGOPS Oper. Syst. Rev.*, 29(5):79–95, 1995.

[PNF07] Juan Piernas, Jarek Nieplocha, and Evan J. Felix. Evaluation of Active Storage Strategies for the Lustre Parallel File System. In *Proceedings of SC07*, November 2007.

[PSK08] Christina M. Patrick, SeungWoo Son, and Mahmut Kandemir. Comparative evaluation of overlap strategies with study of I/O overlap in MPI-IO. volume 42, pages 43–49, New York, NY, USA, 2008. ACM.

[PTH+00] Jean-Pierre Prost, Richard Treumann, Richard Hedges, Alice E. Koniges, and Alison White. Towards a High-Performance Implementation of MPI-IO on Top of GPFS. In *Euro-Par '00: Proceedings from the 6th International Euro-Par Conference on Parallel Processing*, pages 1253–1262, London, UK, 2000. Springer-Verlag.

[PTH+01] Jean-Pierre Prost, Richard Treumann, Richard Hedges, Bin Jia, and Alice Koniges. MPI-IO/GPFS, an optimized implementation of MPI-IO on top of GPFS. In *Supercomputing '01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, pages 17–17, New York, NY, USA, 2001. ACM Press.

[RDED97] Russ Rew, Glenn Davis, Steve Emmerson, and Harvey Davies. NetCDF User's Guide for C - An Access Interface for ..., 1997.

[Rot07] Philip C. Roth. Characterizing the I/O behavior of scientific applications on the Cray XT. In *PDSW '07: Proceedings of the 2nd international workshop on Petascale data storage*, pages 50–55, New York, NY, USA, 2007. ACM.

[SAS08] Hongzhang Shan, Katie Antypas, and John Shalf. Characterizing and predicting the i/o performance of hpc applications using a parameterized synthetic benchmark. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–12, Piscataway, NJ, USA, 2008. IEEE Press.

[SCJ+95] K.E. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett. Server-directed collective I/O in Panda. In *Proceedings of Supercomputing '95*, 1995.

[SGI] SGI MPI home page. *http://www.sgi.com.*

[SH02] F. Schmuck and R. Haskin. GPFS: A Shared-Disk File System for Large Computing Clusters. In *Proceedings of FAST*, 2002.

[SHea06] Yu. H. Sahoo, R.K. Howson, and et all. High performance file I/O for the Blue Gene/L supercomputer. *HPCA*, pages 187–196, 2006.

[SR97] E. Smirni and D.A. Reed. Workload Characterization of I/O Intensive Parallel Applications. In *Proceedings of the Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, June 1997.

[SR98] H. Simitici and D.A. Reed. A Comparison of Logical and Physical Parallel I/O Patterns. In *International Journal of High Performance Computing Applications, special issue (I/O in Parallel Applications), 12(3)*, pages 364–380, 1998.

[Ter] Teragrid home page. *http://www.teragrid.org/.*

[TGL99a] R. Thakur, W. Gropp, and E. Lusk. Data Sieving and Collective I/O in ROMIO. In *Proc. of the 7th Symposium on the Frontiers of Massively Parallel Computation*, pages 182–189, February 1999.

[TGL99b] R. Thakur, W. Gropp, and E. Lusk. On Implementing MPI-IO Portably and with High Performance. In *Proc. of IOPADS*, pages 23–32, May 1999.

[TGZ+04] Hong Tang, Aziz Gulbeden, Jingyu Zhou, William Strathearn, Tao Yang, and Lingkun Chu. A self-organizing storage cluster for parallel data-intensive applications. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 52, Washington, DC, USA, 2004. IEEE Computer Society.

[TL96] Rajeev Thakur and Ewing Lusk. An Abstract-Device Interface for Implementing Portable Parallel-I/O Interfaces. In *Proc. of The 6th Symposium on the Frontiers of Massively Parallel Computation*, pages 180–187, 1996.

[TU99] Hakan Taki and Gil Utard. MPI-IO on a Parallel File System for Cluster of Workstations. In *IWCC '99: Proceedings of the 1st IEEE Computer Society International Workshop on Cluster Computing*, page 150, Washington, DC, USA, 1999. IEEE Computer Society.

[VAD+06] J.S. Vetter, S.R. Alam, Jr. Dunigan, T.H., M.R. Fahey, P.C. Roth, and P.H. Worley. Early evaluation of the Cray XT3. *IPDPS*, pages 10 pp.–, April 2006.

[VHR+08] S.R. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, A. Singh, T. Jacob, S. Jain, V. Erraguntla, C. Roberts, Y. Hoskote, N. Borkar, and S. Borkar. An 80-Tile Sub-100-W TeraFLOPS Processor in 65-nm CMOS. *Solid-State Circuits, IEEE Journal of*, 43(1):29–41, Jan. 2008.

[WBM+06] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: a scalable, high-performance distributed file system. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, pages 307–320, Berkeley, CA, USA, 2006. USENIX Association.

[WdW03] P. Wong and R.F.Van der Wijngaart. NAS Parallel Benchmarks I/O Version 2.4. Technical report, NASA Ames Research Center, 2003.

[Wei06] Wei-keng Liao and Kenin Coloma and Alok Choudhary and Lee Ward and Eric Russell and Neil Pundit. Scalable Design and Implementations for MPI Parallel Overlapping I/O. *IEEE Transactions on Parallel and Distributed Systems*, 17:1264–1276, 2006.

[Wei07] Weikuan Yu and Jeffrey Vetter and R. Shane Canon and Song Jiang. Exploiting Lustre File Joining for Effective Collective IO. In *CCGRID '07: Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid*, pages 267–274, Washington, DC, USA, 2007. IEEE Computer Society.

[WFI+09] Michael Wilde, Ian Foster, Kamil Iskra, Pete Beckman, Zhao Zhang, Allan Espinosa, Mihael Hategan, Ben Clifford, and Ioan Raicu. Parallel Scripting for Applications at the Petascale and Beyond. *Computer*, 42(11):50–60, 2009.

[WLBM07] Sage A. Weil, Andrew W. Leung, Scott A. Brandt, and Carlos Maltzahn. Rados: a scalable, reliable storage service for petabyte-scale storage clusters. In Garth A. Gibson, editor, *PDSW*, pages 35–44. ACM Press, 2007.

[WSC+96] M. Winslett, K.E. Seamons, Y. Chen, Y. Cho, S. Kuo, and M. Subramaniam. The Panda library for parallel I/O of large multidimensional arrays. In *Proceedings of Scalable Parallel Libraries Conference III*, October 1996.

[WXH+04] Feng Wang, Qin Xin, Bo Hong, Scott A. Br, Ethan L. Miller, Darrell D. E. Long, and Tyce T. Mclarty. File system workload analysis for large scale scientific computing applications. In *In Proceedings of the 21st IEEE / 12th NASA Goddard Conference on Mass Storage Systems and Technologies*, pages 139–152, 2004.

[YV08] Weikuan Yu and Jeffrey Vetter. ParColl: Partitioned Collective I/O on the Cray XT. *ICPP*, pages 562–569, 2008.

[YVC07a] Weikuan Yu, Jeffrey S. Vetter, and R. Shane Canon. OPAL: An Open-Source MPI-IO Library over Cray XT. In *SNAPI '07*, pages 41–46, 2007.

[YVC07b] Weikuan Yu, Jeffrey S. Vetter, and R. Shane Canon. OPAL: An Open-Source MPI-IO Library over Cray XT. volume 0, pages 41–46, Los Alamitos, CA, USA, 2007. IEEE Computer Society.